# AFIPS CONFERENCE PROCEEDINGS

VOLUME 35

1969

FALL JOINT COMPUTER CONFERENCE

November 18 - 20, 1969
Las Vegas, Nevada

AFIPS PRESS
210 SUMMIT AVENUE
MONTVALE, NEW JERSEY 07645

Library of Congress Catalog Card Number 55–44701

AFIPS PRESS
210 Summit Avenue
Montvale, New Jersey 07645

Printed in the United States of America

# CONTENTS

# A survey of techniques for recognizing parallel processable streams in computer programs*

*by* C. V. RAMAMOORTHY and M. J. GONZALEZ

*The University of Texas*
Austin, Texas

## INTRODUCTION

State-of-the-art advances—in particular, anticipated advances generated by LSI—have given fresh impetus to research in the area of parallel processing. The motives for parallel processing include the following:

1. Real-time urgency. Parallel processing can increase the speed of computation beyond the limit imposed by technological limitations.
2. Reduction of turnaround time of high priority jobs.
3. Reduction of memory and time requirements for "housekeeping" chores. The simultaneous but properly interlocked operations of reading inputs into memory and error checking and editing can reduce the need for large intermediate storages or costly transfers between members in a storage hierarchy.
4. An increase in simultaneous service to many users. In the field of the computer utility, for example, periods of peak demand are difficult to predict. The availability of spare processors enables an installation to minimize the effects of these peak periods. In addition, in the event of a system failure, faster computational speeds permit service to be provided to more users before the failure occurs.
5. Improved performance in a uniprocessor multiprogrammed environment. Even in a uniprocessor environment, parallel processable segments of high priority jobs can be overlapped so that when one segment is waiting for I/O, the processor can be computing its companion segment. Thus an overall speed up in execution is achieved.

With reference to a single program, the term "parallelism" can be applied at several levels. Parallelism within a program can exist from the level of statements of procedural languages to the level of micro operations. Throughout this paper, discussion will be confined to the more general "task" parallelism. The term "task" (process) generally is intended to mean a self-contained portion of a computation which once initiated can be carried out to its completion without the need for additional inputs. Thus the term can be applied to a single statement or a group of statements.

In contrast to the way the term "level" was used above, task parallelism can exist at several levels within a hierarchy of levels. The statements of the main program of a FORTRAN program, for example, are said to be tasks of the first level. The statements within a subroutine called by the main program would then be second level tasks. If this subroutine itself called another subroutine, then the statements within the latter subroutine would be of the third level, etc. Thus a sequentially organized program can be represented by a hierarchy of levels as shown in Figure 1. Each

* This work was supported by NASA Grant NGR 44-012-144.

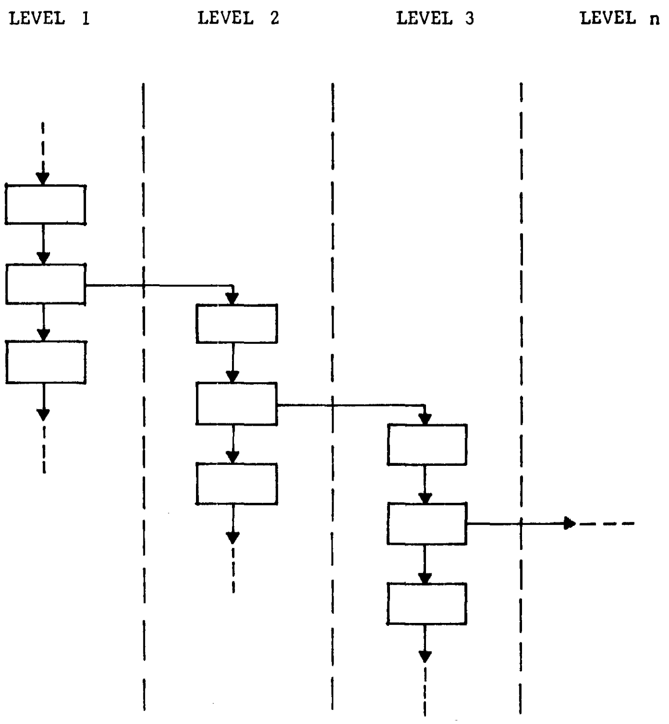


Figure 1—Hierarchical representation of a sequentially organized program

block within a level represents a single task; as before, a task can represent a statement or a group of statements.

Once a sequentially organized program is resolved into its various levels, a fundamental consideration of parallel processing becomes prominent—namely that of recognizing tasks within individual levels which can be executed in parallel. Assuming the existence of a system which can process independent tasks in parallel, this problem can be approached from two directions. The first approach provides the programmer with additional tools which enable him to explicitly indicate the parallel processable tasks. If it is decided to make this indication independent of the programmer, then it is necessary to recognize the parallel processable tasks implicitly by analysis of the relationship between tasks within the source program.

After the information is obtained by either of these approaches, it must still be communicated to and utilized by the operating system. At this point, efficient resource utilization becomes the prime consideration.

The conditions which determine whether or not two tasks can be executed in parallel have been investigated by Bernstein.[1] Consider several tasks, $T_i$, of a sequentially organized program illustrated by a flow chart as shown in Figure 2(a). If the execution of task $T_3$ is independent of whether tasks $T_1$ and $T_2$ are executed sequentially as shown in Figure 2(a) or 2(b), then parallelism is said to exist between tasks $T_1$ and $T_2$. They can, therefore, be executed in parallel as shown in Figure 2(c).

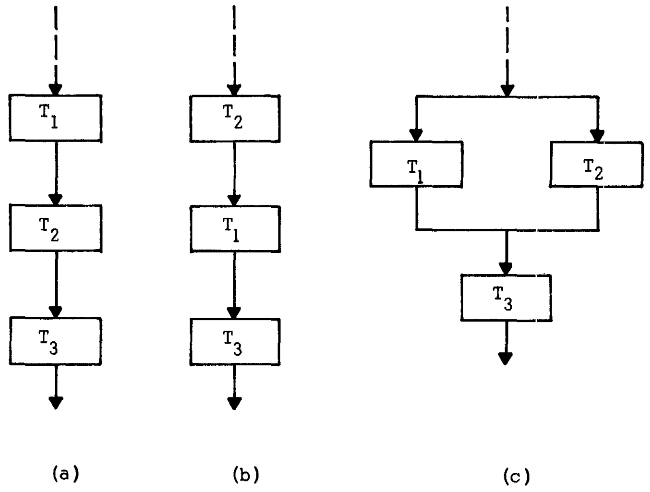


Figure 2—Sequential and parallel execution of a computational process

This "commutativity" is a necessary but not sufficient condition for parallel processing. There may exist, for instance, two processes which can be executed in either order but not in parallel. For example, the inverse of a matrix $A$ can be obtained in either of the two ways shown below.

| (1) | (2) |
|---|---|
| a) Obtain transpose of $A$ | a) Obtain matrix of cofactors of $A$ |
| b) Obtain matrix of cofactors of the transposed matrix | b) Transpose matrix of cofactors |
| c) Divide result by determinant of $A$ | c) Divide result by determinant of $A$ |

Thus obtaining the matrix of cofactors and the transposition operation are two distinct processes which can be executed in alternate order with the same result. They cannot, however, be executed in parallel.

Other complications may arise due to hardware limitations. Two tasks, for example, may need to access the same memory. In this and similar situations, requests for service must be queued. Djkstra, Knuth, and Coffman[2,3,4] have developed efficient scheduling procedures for using common resources.

In terms of sets representing memory locations, Bernstein has developed the conditions which must be satisfied before sequentially organized processes can be executed in parallel. These are based on four separate ways in which a sequence of instructions can use a memory location:

(1) The location is only fetched during the execution of $T_i$.

(2) The location is only stored during the execution of $T_i$.

(3) The first operation within a task involves a fetch with respect to a location; one of the succeeding operations of $T_i$ stores in this location.

(4) The first operation within a task involves a store with respect to a location; one of the succeeding operations of $T_i$ fetches this location.

Assuming a machine model in which processors are allowed to communicate directly with the memory and multi-access operations are permitted, the conditions for strictly parallel execution of two tasks or program blocks can be stated as follows.

(1) The areas of memory which Task 1 "reads" and onto which Task 2 "writes" should be mutually exclusive, and vice-versa.

(2) With respect to the next task in a sequential process, Tasks 1 and 2 should not store information in a common location.

The conditions listed by Bernstein are sufficient to guarantee commutativity and parallelism of two program blocks. He has shown, however, that there do not exist algorithms for deciding the commutativity or parallelism of arbitrary program blocks.

As an example of what has been discussed here consider the tasks shown below which represent FORTRAN statements for evaluation of three arithmetic expressions.

$$X = (A+B) * (A-B)$$
$$Y = (C-D) / (C+D)$$
$$Z = X+Y$$

Because the execution of the third expression is independent of the order in which the first two expressions are executed, the first two expressions can be executed in parallel.

Parallelism within a task can also exist when individual components of compound tasks can be executed concurrently. In the same manner that individual processors can be assigned to independent tasks, individual functional units can be assigned to independent components within a task. The motivation remains the same—a decrease in execution time of individual tasks. The CDC 6600, for example, can utilize several arithmetic units to perform several operations simultaneously. This type of parallelism can be illustrated by the arithmetic expression which follows.

$$X = (A+B) * (C-D)$$

Normally, this expression would be evaluated in a manner similar to that shown in Figure 3(a). The independent components within the expression, however, permit parallel execution as shown in Figure 3(b) with the same results.

*Explicit and implicit parallelsim*

In the explicit approach to parallelism, the programmer himself indicates the tasks within a computational process which can be executed in parallel. This is normally done by means of additional instructions in the programming language. This approach can be illustrated by the techniques described by Conway, Opler, Gosden, and others[5,6,7]. FORK in the FORK and JOIN technique[5] indicates thep arallel processability of a specified set of tasks within a process. The next sequence of tasks will not be initiated until all

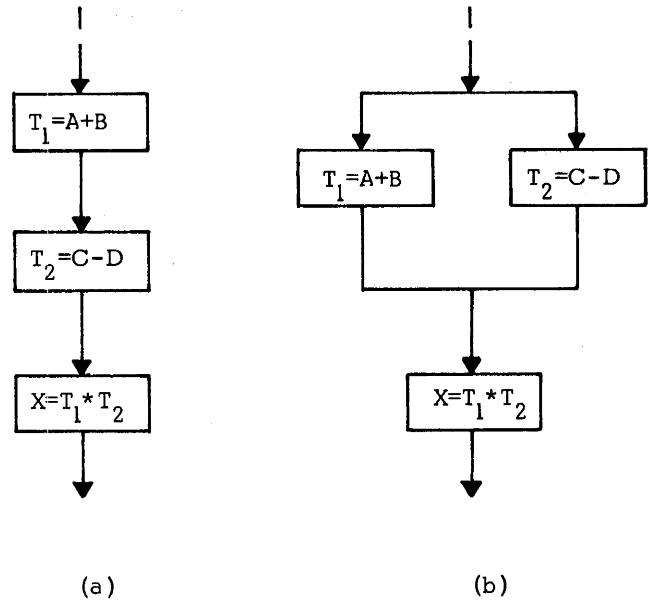


Figure 3—Illustration of parallelism within a compound task

the tasks emanating from a FORK converge to a JOIN statement.

In some instances, some of the parallel operations initiated by the FORK instruction do not have to be completed before processing can continue. For example, one of these branch operations may be designed to alert an I/O unit to the fact that it is to be used momentarily. The conventional FORK must be modified to take care of these situations. Execution of an IDLE statement, for example, permits processors to be released without initiation of further action.[7] The FORK and JOIN TECHNIQUE is illustrated in Figure 4.

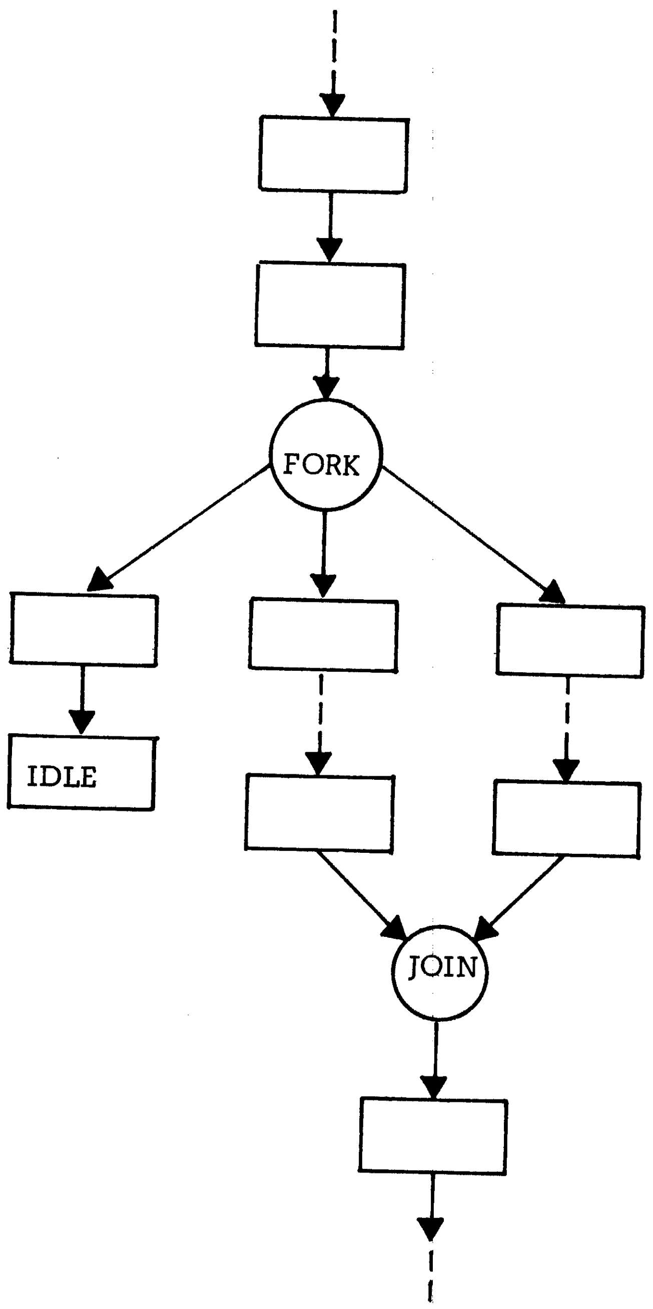


Figure 4—FORK and JOIN technique

Another example of the explicit approach is the PARALLEL FOR[7] which takes advantage of parallel operations generated by the FOR statement in ALGOL and similar constructs in other languages. For example, the sum of two $n \times n$ matrices consists essentially of $n^2$ independent operations. If n processors were available, the addition process could be organized such that entire rows or columns could be added simultaneously. Thus the addition of the two matrices could be accomplished in n units of time. Another example of this approach is the programming language PL/1 which provides the TASK option with the CALL statement which indicates concurrent execution of parallel tasks.

An additional way of indicating parallelism explicitly is to write a language which exploits the parallelism in algorithms to be implemented by the operating system. This is the case with TRANQUIL,[8,21] an ALGOL-like language to be utilized by the array processors of the ILLIAC IV. The situation is unique in that the language was created after a system was devised to solve an existing problem. "The task of compiling a language for the ILLIAC IV is more difficult than compiling for conventional machines simply because of the different hardware organization and the need to utilize its parallelism efficiently." A limitation of this approach is that programs written in that particular language can only be run on array-type computers and is, therefore, heavily machine dependent.

The implicit approach to parallelism does not depend on the programmer for determination of inherent parallelism but relies instead on indicators existing within the program itself. In contrast to the relative ease of implementation of explicit parallelism, the implicit approach is associated with complex compiling and supervisory programs.

The detection of inherent parallelism between a set of tasks depends on thorough analysis of the source program using Bernstein's conditions. Implementation of a recognition scheme to accomplish this detection is dependent on the source language. Thus a recognizer which is universally applicable cannot be implemented.

An algorithm developed by Fisher[9] approaches the problem of parallel task detection in a general manner. His algorithm utilizes the input and output sets of each task (process) to determine essential ordering and thus inherent parallelism. Given such information as the number of processes to be analyzed, the input and output set for each process, the given permissible ordering among the processes, and any initially known essential order among the processes, the algorithm generates the essential serial ordering relation and the covering for the essential serial ordering relation. This covering provides an indication of the tasks within the overall process which can be executed concurrently.

Basically, this work formalizes in the form of an algorithm the conditions for parallel processing developed by Bernstein. The conditions for parallel processing between two tasks are extended to an overall process

## Detection of task parallelism—A new approach

The next subject covered in this paper involves implicit detection of parallel processable tasks within programs prepared for serial execution. An indication is desired of the tasks which can be executed in parallel and the tasks which must be completed before the start of the next sequence of tasks. Thus the problem can be broken down in two parts—recognizing the relationships between tasks within a level and using this information to indicate the ordering between tasks.

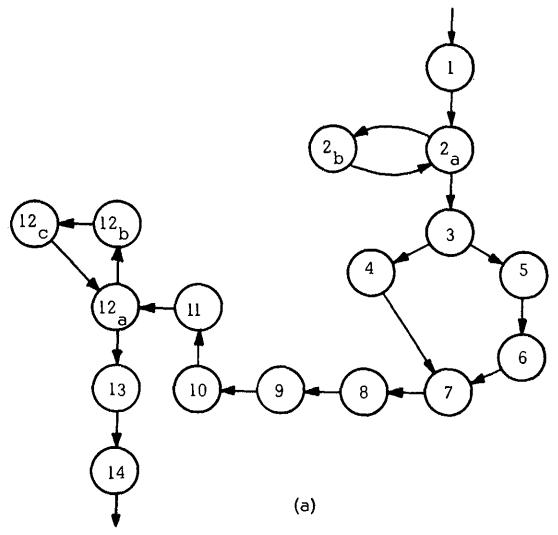
The approach presented here is based on the fact that computational processes can be modeled by oriented graphs in which the vertices (nodes) represent single tasks and the oriented edges (directed branches) represent the permissible transition to the next task in sequence. The graph (and thus the computational process) can be represented in a computer by means of a Connectivity Matrix, C.[10,11] $C$ is of dimension $n \times n$ such that $C_{ij}$ is a "1" if and only if there is a directed edge from node i to node j, and it is "0" otherwise. The properties of the directed graph and hence of the computational process it represents can be studied by simple manipulations of the connectivity matrix.

A graph consisting of a set of vertices is said to be *strongly connected* if and only if any node in it is reachable from any other. A *subgraph* of any graph is defined as consisting of a subset of vertices with all the edges between them retained. A *maximal strongly connected* (M.S.C.) *subgraph* is a strongly connected subgraph that includes all possible nodes which are strongly connected with each other. Given a connectivity matrix of a graph, all its M.S.C. subgraphs can be determined simply by well-known methods.[10] A given program graph can be *reduced* by replacing each of its M.S.C. subgraphs by a single vertex and retaining the edges connected between these vertices and others. After the reduction, the *reduced graph* will not contain any strongly connected components.

The paragraphs which follow will describe the sequence of operations needed to prepare for parallel processing in a multiprocessor computer a program written for a uniprocessor machine.

(1) The first step is to derive the *program graph* which identifies the sequence in which the computation al tasks are performed in the sequentially codeprogram. Figure 5(a) illustrates an example program graph. The program graph is represented in the computer by its connectivity matrix. The connectivity matrix for the example is given in Figure 5(b).

(2) By an analysis of the connectivity matrix, the maximal strongly connected subgraphs are determined by simple operations.[10] This type of subgraph is illustrated by tasks 2 and 12 in Figure 5. Each M.S.C. subgraph is next considered as a single task, and the graph, called the reduced graph, is derived. The reduced graph does not contain any loops or strongly



(a)

$$
\begin{array}{c|cccccccccccccccccc}
 & 1 & 2_a & 2_b & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12_a & 12_b & 12_c & 13 & 14 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
2_a & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
2_b & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
12_a & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
12_b & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
12_c & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
13 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
14 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

(b)

Figure 5—Program graph of a serially coded program and its connectivity matrix

connected elements. In this graph, when two or more edges emanate from a vertex, a conditional branching is indicated. That is, the execution sequence will take only one of the indicated alternatives. A vertex which initiates the branching operation will be called a *decision* or *branch* vertex. The reduced graph for the example program graph is shown in Figure 6. In this graph, vertex 3 represents a branch vertex.

(3) The next step is to derive the final program graph and its connectivity matrix *T*. The elements of *T* are obtained by analyzing the inputs of each vertex in the reduced graph. An element, $T_{ij}$, is a "1" if and only if the j-th task (vertex) of the reduced graph has as one of its inputs the output of task i; otherwise $T_{ij}$ is a "0". Figure 7 illustrates the final program for the example after consideration is given to the input-output relationships of each task. The connectivity matrix for the final program graph is shown in Figure 8.

From the sufficiency conditions for task parallelism, two tasks can be executed in parallel if the input set of one task does not depend on the output set of the other and vice versa. The technique outlined in Step 4 detects this relationship and uses it to provide an ordering for task execution.

(4) The vertices of the final program graph are

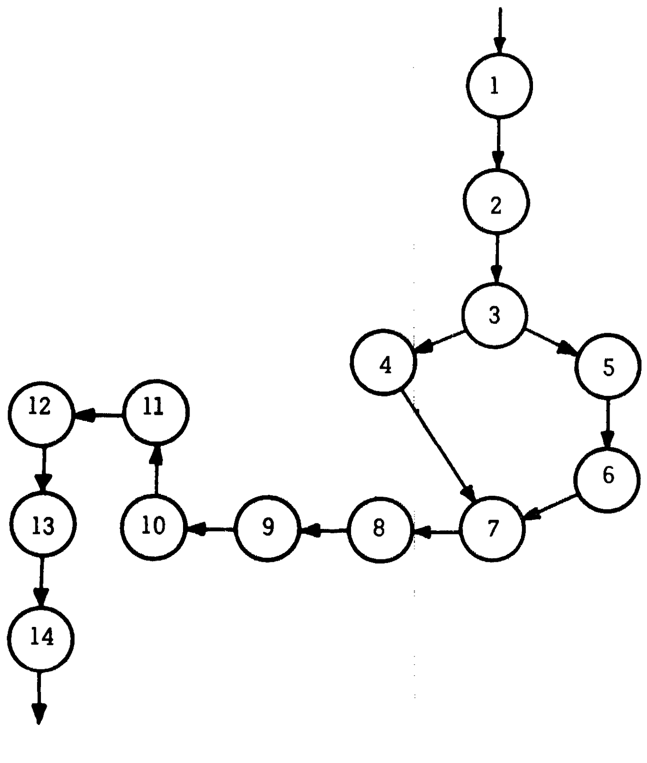


Figure 6—Reduced program graph of the serially coded program

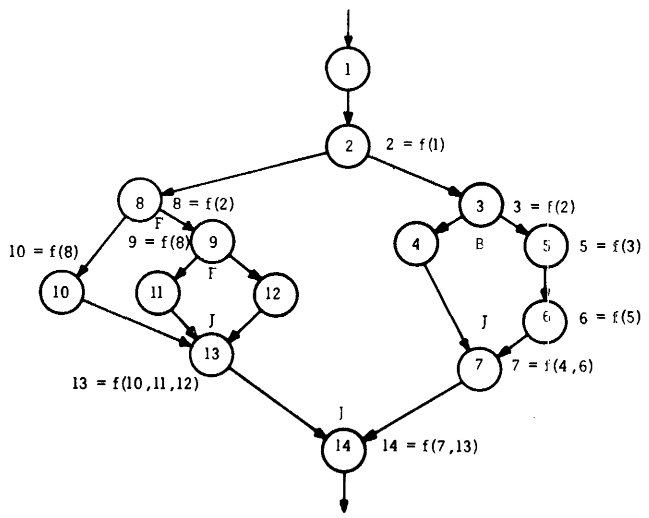


Figure 7—Final program graph of the parallel processable program

$$\underline{T} = \begin{array}{c|cccccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\hline
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
13 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
14 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

Precedence Partitions {1} , {2} , {3,8} , {4,5,9,10}
{6,11,12}, {7,13}, {14}

Figure 8—Connectivity matrix of the final program graph

partitioned into "precedence partitions"[12] as follows. Using the connectivity matrix *T*, a column (or columns) containing only zeroes is located. Let this column correspond to vertex $v_1$. Next delete from *T* both the column and the row corresponding to this vertex. The first precedence partiton is $P_1 = \{v_1\}$. Using the remaining portion of T, locate vertices $\{v_{21}, v_{22}, \ldots\}$ which correspond to columns containing only zeroes. The second precedence partition $P_2$ thus contains vertices $\{v_{21}, v_{22}, \ldots\}$. This implies that tasks in set $P_2 =$

$\{v_{21}, v_{22},\ldots\}$ can be initiated and executed in parallel after the tasks in the previous partition (i.e., $P_1$) have been completed. Next delete from $T$ the columns and rows corresponding to vertices in $P_2$. This procedure is repeated to obtain precedence partitions $P_3,P_4,\ldots P_p$, until no more columns or rows remain in the $T$ matrix. It can be shown that this partitioning procedure is valid for connectivity matrices of graphs which contain no strongly connected components.

The implication of this precedence partitioning is that if $P_1,P_2,\ldots P_p$ corresponds to times $t_1,t_2,\ldots t_p$, the earliest time that a task in partition $P_i$ can be initiated is $t_i$.

The final program graph contains the following types of vertices: (1) The branch or decision type vertex from which the execution sequence selects a task from a set of alternative tasks. (2) The Fork vertex which can initiate a set of parallel tasks. (3) The Join vertex to which a set of parallel tasks converge after their execution. (4) The normal vertex which receives its input set from the outputs of preceding tasks. Figure 7a indicates the final program graph with the first three types of vertices indicated by B, F, and J, respectively.

(5) From precedence partitioning and the final program graph, a Task Scheduling Table can be developed. This table, shown in Table I, serves as an input to the operating system to help in the scheduling of tasks. For example, if the task being executed is a Fork task, a look-ahead feature of the system can prepare for parallel execution of the tasks to be initated upon completion of the currently active task.

(6) The precedence partitions of Step 4 provide an indication of the *earliest* time at which a task may be initiated. It is also desirable, however, to provide an indication of the *latest* time at which a task may be initiated. This information can be obtained by performing precedence partitions on the transpose of the $T$ matrix. This process can be referred to as "row partitions". The implication here is that if task is in the partition corresponding to time period $t_k$, then $t_k$ is the latest time that the task i can be initiated.

Using both the row and column partitions, the permissible initiation time for each task can be derived as shown in Table II. Task 4, for example, can be initiated during $t_4$ or $t_5$ depending on the availability of processors.

At this point it is desirable to clarify some possible misinterpretations of the implications of this method. The method presented here does not try to determine whether any or all of the iterations within a loop can be executed simultaneously. Rather the iterations executed sequentially are considered as a single task.

TABLE I—Task scheduling table

| TIME | INPUTS TO TASKS | TASK NUMBER | TASK TYPE |
|---|---|---|---|
| $t_1$ | - | 1 | |
| $t_2$ | 1 | 2 | FORK |
| $t_3$ | 2 | 3 | BRANCH |
| $t_3$ | 2 | 8 | FORK |
| $t_4$ | 3 | 4 | |
| $t_4$ | 3 | 5 | |
| $t_4$ | 8 | 9 | FORK |
| $t_4$ | 8 | 10 | |
| $t_5$ | 5 | 6 | |
| $t_5$ | 9 | 11 | |
| $t_5$ | 9 | 12 | |
| $t_6$ | 4,6 | 7 | JOIN |
| $t_6$ | 10,11,12 | 13 | JOIN |
| $t_7$ | 7,13 | 14 | JOIN |

For this reason, the undecidability problem introduced by Bernstein is not a factor here.

In addition, precedence partitions may place the successors of a conditional within the same partition. The interpretation of this is that only one of the successors will be executed, and it can be executed in parallel with the other tasks within that partition.

## The FORTRAN parallel task recognizer

In order to determine the degree of applicability of the method described above, it was decided to apply the method to a sample FORTRAN program. This was accomplished by writing a program whose input consists of a FORTRAN source program; its output consists of a listing of the tasks within the *first level* of the source program which can be executed in parallel. The program written to accomplish this parallel task

TABLE II—Permissible task initiation time

| COLUMN PARTITIONS | |
|---|---|
| TIME | TASK |
| $t_1$ | 1 |
| $t_2$ | 2 |
| $t_3$ | 3,8 |
| $t_4$ | 4,5,9,10 |
| $t_5$ | 6,11,12 |
| $t_6$ | 7,13 |
| $t_7$ | 14 |
| **ROW PARTITIONS** | |
| $t_1$ | 1 |
| $t_2$ | 2 |
| $t_3$ | 3,8 |
| $t_4$ | 5,9 |
| $t_5$ | 4,6,10,11,12 |
| $t_6$ | 7,13 |
| $t_7$ | 14 |

| PERMISSIBLE TASK INITIATION PERIODS | |
|---|---|
| TASK | TIME |
| 1 | $t_1$ |
| 2 | $t_2$ |
| 3 | $t_3$ |
| 4 | $t_4$,$t_5$ |
| 5 | $t_4$ |
| 6 | $t_5$ |
| 7 | $t_6$ |
| 8 | $t_3$ |
| 9 | $t_4$ |
| 10 | $t_4$,$t_5$ |
| 11 | $t_5$ |
| 12 | $t_5$ |
| 13 | $t_6$ |
| 14 | $t_7$ |

detection is known in its final form as a FORTRAN Parallel Task Recognizer.[13]

The recognizer, also written in FORTRAN, relies on indicators generated by the way in which the program is actually written. Consider the expressions given below.

$$X1 = f_1(A,B)$$
$$X2 = f_2(C,D)$$

Because the right-hand side of the second expression does not contain a parameter generated by the computation which immediately precedes it, the two expressions can be executed in parallel. If, on the other hand, the expressions were rewritten as shown below, the termination of the first computation would have to precede the initiation of the second.

$$X1 = f_1(A,B)$$
$$X2 = f_2(X1,C)$$

The recognizer performs this determination by comparing the parameters on the right-hand of the equality sign to outcomes generated by previous statements.

Other FORTRAN instructions can be analyzed similarly. Consider the arithmetic IF:

IF (X − Y) 3,4,5

Here the parameters within the parentheses must be compared to the outputs of preceding statements in order to determine essential order.

Other FORTRAN instructions are analyzed in a similar manner in order to generate the connectivity matrix for the source program. During this analysis the recognizer assigns numbers to the executable statements of the source program. After this is completed, the recognizer proceeds with the method of precedence partitions described earlier. Precedence partitions yield a list of blocks which contain the statement numbers which can be executed concurrently.

Figure 9 shows a block diagram of the steps taken by the recognizer to generate the parallel processable tasks within the first level of a FORTRAN source program.

Some statements within the FORTRAN set are treated somewhat differently. The DO statement, for example, does not itself contain any input or output parameters but instead generates a series of repeated operations. Because of the loop considerations mentioned earlier, and because the rules of FORTRAN require entrance into a loop only through the DO statement, all the statements contained within a DO loop are considered as a single task. A loop, however, may contain a large number of statements, and a great amount of potential parallelism may be lost if consideration is not given to the statements within the loop. For this reason, the recognizer generates a separate connectivity matrix for each DO loop within the program.

The recognizer itself possesses limitations which must be eliminated before it can be applied to programs of a complex nature. For example, only a subset of the entire FORTRAN set is considered for recogniton. This could be corrected by expanding the recognition process to include a more complete set of instructions.

In addition to the DO statement, loops can also be

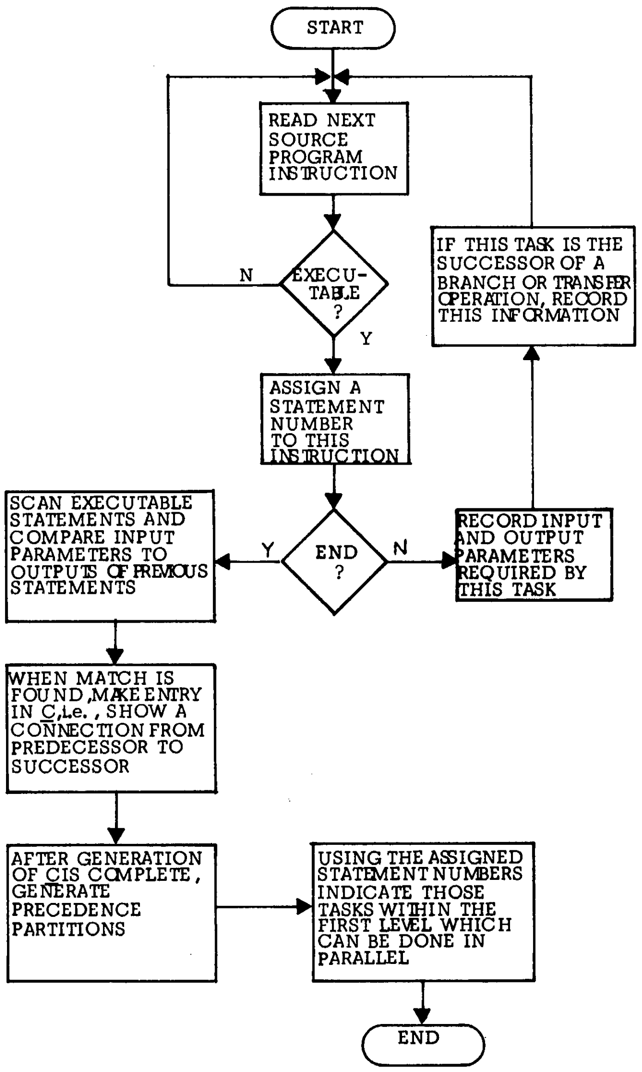

START
READ NEXT SOURCE PROGRAM INSTRUCTION
EXECU-TABLE ?
N
Y
IF THIS TASK IS THE SUCCESSOR OF A BRANCH OR TRANSFER OPERATION, RECORD THIS INFORMATION
ASSIGN A STATEMENT NUMBER TO THIS INSTRUCTION
END ?
Y
N
SCAN EXECUTABLE STATEMENTS AND COMPARE INPUT PARAMETERS TO OUTPUTS OF PREVIOUS STATEMENTS
RECORD INPUT AND OUTPUT PARAMETERS REQUIRED BY THIS TASK
WHEN MATCH IS FOUND, MAKE ENTRY IN C, i.e., SHOW A CONNECTION FROM PREDECESSOR TO SUCCESSOR
AFTER GENERATION OF CIS COMPLETE, GENERATE PRECEDENCE PARTITIONS
USING THE ASSIGNED STATEMENT NUMBERS INDICATE THOSE TASKS WITHIN THE FIRST LEVEL WHICH CAN BE DONE IN PARALLEL
END

Figure 9—Block diagram of the FORTRAN parallel task recognizer

```
    C        THIS IS A TEST PROGRAM DESIGNED TO CHECK PPS
             DIMENSION A1(10),A2(10),A3(10)
             INTEGER A1,A2,ABC,A2,X2,B,C,D
 1           READ 100,  (A1(I),I=1,10),B,C,D
 2           READ 100,  (A2(I),I=1,10),NS,NST,NSTU
 3           DO 10  I=1,10
 4           IF(A1(I)-A2(I))20,30,40
 5       20  X1=(A1(I))*(B-C)
 6           X2=D+(B/C)
 7           A3(I)=X1*X2
 8       10  CONTINUE
    C        THIS IS A TEST COMMENT
 9       30  PRINT  200,B,C,D
10       40  CALL ALPHA(A1,A2,ABC,B4,B5)
11           PRINT 3057,X1,X2,(A3(I),I=1,10)
12           CALL BETA(X1,X2,A3,B6)
13           IF(B4-B5)50,50,60
14       50  READ  315,E,F,G,H
15           X3=(E*F)+(G-H)
16           X4=B6+G                                             PARALLEL
17           X5=X3-X4                                            PROCESSABLE
18           X6=(B4+B5)*X5                                       TASKS
19       60  PRINT  4,X3,X4,X5                                        (1,2)
20           PRINT  52,(A1(I),I=1,10),ABC,C,(A3(I),I=1,10)            (3)
        100  FORMAT(10I2,3I3)                                         (9,10,11,12)
        200  FORMAT(1H0,8  B  C  D*,/,3I3)                            (13)
       3057  FORMAT(1H  ,2I3,10F7.1)                                  (14)
        315  FORMAT(4F7.4)                                            (15,16)
          4  FORMAT(3F7.4)                                            (17)
         52  FORMAT(12I3,10F7.1)                                      (18,19,20)
21           END

                            (a)                                       (b)
```

Figure 10—An example of the recognition process.

created by branch and transfer operations such as the IF and GO TO instructions. To eliminate these loops, it would be necessary to analyze the connectivity matrix in the manner mentioned earlier before beginning the process of precedence partitions. The recognizer does not presently perform this analysis.

Nested DO loops are not permitted, and the source program size is limited in the number of executable statements it may have and in the number of parameters any one statement can contain.

Some of these limitations could be eliminated quite easily; others would require a considerable amount of effort. To allow a source program of arbitrary size would require a somewhat more elaborate handling of memory requirements and associated problems. At the present time the recognizer consists of a main program and six subroutines. In its present form the recognizer consists of approximately 1300 statements.

The recognizer is presently written in such a manner that it will detect only first level parallelism. The method it uses, however, can be applied to parallelism at any level.

The theory of operation of the FORTRAN parallel task recognizer will be illustrated by applying the recognition techniques to a sample FORTRAN program. Figure 10(a) is a listing of the sample program showing the individual tasks. Figure 10(b) is a listing of the parallel processable tasks as determined by precedence partitions. The numbers to the left of the executable statements are the numbers assigned by the recognizer during the recognition phase.

Elimination of the limitations mentioned here and other limitations not mentioned explicitly will be the subject of future effort.

## Observations and comments

Regardless of the manner in which the subject of parallel processing is approached, common problems arise. Prominent among these is a need to protect common data. If two tasks are considered for concurrent execution and one task accesses a memory location and the other amends it, then strict observance must be paid to the order in which this is done. The

FORTRAN recognizer, for example, may determine that two subroutines can be executed in parallel. At the present time no consideration is given to the fact that both subroutines may access common data through COMMON or EQUIVALENCE statements.

In order to truly optimize execution time for a program which is set up for parallel processing, it would be highly desirable to determine the time required for execution of the individual tasks within the process. It is not enough to merely determine that two tasks can be executed concurrently; the primary goal is that this parallel execution result in higher resource utilization and improved throughput. If the time required for the execution of one task is 100 times that of the other, for example, then it may be desirable to execute the two tasks serially rather than in parallel. The reasoning here is that no time would be spent in allocating processors and so forth.

Determination of task execution time, however, is not a simple matter. Exhaustive measurements of the type suggested by Russell and Estrin[14] would provide the type of information mentioned here.

Another problem area involves implementation of special purpose languages such as TRANQUIL. It was mentioned earlier that programs written in a language of this type are highly machine-limited. It would be highly desirable to be able to implement programs written in these languages in systems which are not designed to take advantage of parallelism. Along these lines, the programming generality suggested by Dennis[15] may be significant.

It should be pointed out that all the techniques which have been discussed here will create a certain amount of overhead. For this reason it is felt that a parallel task recognizer, for example, would be best suited for implementation with production programs. Thus even though some time would be lost initially, in the long run parallel processing would result in a significant net gain.

## Conclusions

The method of indicating parallel processable tasks introduced here and illustrated in part by the FORTRAN Parallel Recognizer appears to provide enough generality that it is independent of the language, the application, the mode of compilation, and the number of processors in the system. It is anticipated that this method will remain as the basis for further effort in this area.

In addition to the comments made earlier, some possible future areas of effort include determination of possible parallelism of individual iterations within a loop. It is hoped that additional information can be provided to the operating system other than a mere indication of the tasks which can be executed in parallel. This would include the measurements mentioned earlier and an indication of the frequency of execution of individual tasks.

It is also hoped that a sub-language may be developed which can be added to existing languages to assist in the recognition process and the development of recognizer code.

## Detection of parallel components within compound tasks

Several algorithms exist for the detection of independent components within compound tasks.[16,17,18,19] These algorithms are concerned primarily with detection of this type of parallelism within arithmetic expressions. The first three algorithms referenced above are summarized in [19] where a new algorithm is also introduced.

The arithmetic expression which will be used as an example for each algorithm is given below.

$$A + B + C + D * E * F + G + H$$

Throughout this discussion the usual precedence between operators will apply. In order of increasing precedence, the precedence between operators will be as follows: $+$ and $-$, $*$ and/, and $\uparrow$, where $\uparrow$ stands for exponentiation.

### Hellerman's algorithm

This algorithm assumes that the input string is written in reverse Polish notation and contains only binary operators. The string is scanned from left to right replacing by temporary results each occurrence of adjacent operands immediately followed by an operator. These temporary results will be considered as operands during the next passes. Temporary results generated during a given pass are said to be at the same level and therefore can be executed in parallel. There will be as many passes as there are levels in the syntactic tree. The compilation of the expression listed above is shown in Figure 11.

Although this algortihm is simple and fast, it has two shortcomings. The first is a possible difficulty in implementation since it requires the input string to be in Polish notation; the second is its inability to handle operators which are not commutative.

| i | INPUT STRING AFTER THE ith PASS | TEMPORARY RESULTS GENERATED DURING ith PASS |
|---|---|---|
| 0 | AB+C+DE*F*+G+H+ | |
| 1 | R1 C+R2 F*+G+H+ | R1=A+B R2=D*E |
| 2 | R3 R4+G+H+ | R3=R1+C R4=R2*F |
| 3 | R5 G+H+ | R5=R3+R4 |
| 4 | R6 H+ | R6=R5+G |
| 5 | R7 | R7=R6+H |

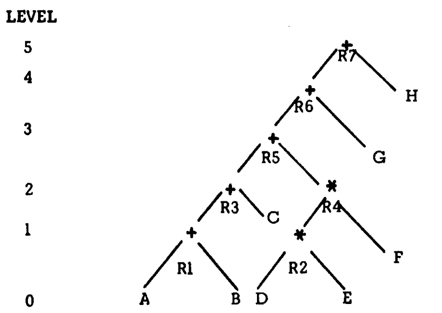


Figure 11—Parallel computation of A+B+C+D*E*F+G+H using Hellerman's algorithm

## Stone's algorithm

The basic function of this algorithm is to combine two subtrees of the same level into a level that is one higher. For example, A and B, initially of level 0, are combined to form a subtree of level 1. The algorithm then searches for another subtree of level 1 by attempting to combine C and D. Since precedence relationships between operators prohibit this combination, the level of subtree (A+B) is incremented by one. The algorithm now searches for a subtree of level 2 by attempting to combine C, D, and E. Since this combination is also prohibited, subtree (A+B) is incremented to level 3. The next search is successful, and a subtree of level 3 is obtained by combining C, D, E and F. These two subtrees are then combined to form a single subtree of level 4.

In a similar manner the subtree (G+H), originally of level 1, is successively incremented until it achieves a level of 4; at that time it is combined with the other subtree of the same level to form a final tree of level 5.

The algorithm yields an output string in reverse Polish which does not expressly show which operations can be performed in parallel. Even though the output string is generated in one pass, the recursiveness of the algorithm causes it to be slow, and at least one additional pass would be required to specify parallel computations.

## Squire's algorithm

The goal of this algorithm is to form quintuples of temporary results of the form:

Ri (operand 1, operator, operand 2, start level = max [end level op. 1; end level op. 2], end level = start level + 1).

All temporary results which have the same start level can be computed in parallel. Initially, all variables have a start and end level equal to zero.

Scanning begins with the rightmost operator of the input string and proceeds from right to left until an operator is found whose priority is lower than that of the previously scanned operator. In the example the scan would yield the following substring:

D*E*F+G+H

Now a left to right scan proceeds until an operator is found whose priority is lower than that of the leftmost operator of the substring. This yields: D*E*F. At this point a temporary result R1 is available of the form:

R1(D,*,E,0,1).

The temporary result, R1, replaces one of the operands and the other is deleted together with its left operator The new substring is then:

R1*F+G+H.

The left to right scans are repeated until no further qunituple can be produced, and at that time, the right to left scan is re-initiated. The results of the process are shown in Figure 12.

Although the example shows the algorithm applied to an expression containing only binary operators, the algorithm can also handle subtraction and division with a corresponding increase in complexity.

A significant feature of this algorithm is that Polish notation plays no part in either the input string or the output quintuples. Because of the many scans and comparisons the algorithm requires, it becomes more complex as the length of the expression and the diversity of operators within the expression increase.

INITIAL STRING: A+B+C+D*E*F+ G+H

| RIGHT TO LEFT SCAN | LEFT TO RIGHT SCAN |
|---|---|
| D*E*F+G+H | R1*F+G+H |
| | R2+G+H |
| A+B+C+R2+G+H | R3+C+R2+G+H |
| | R4+R3+R2+H |
| | R4+R5+R2 |
| | R6+R2 |
| | R7 |

| QUINTUPLES | Op.1 | OPERATOR | Op.2 | START | END |
|---|---|---|---|---|---|
| R1 | D | * | E | 0 | 1 |
| R2 | F | * | R1 | 1 | 2 |
| R3 | A | + | B | 0 | 1 |
| R4 | C | + | G | 0 | 1 |
| R5 | H | + | R3 | 1 | 2 |
| R6 | R4 | + | R5 | 2 | 3 |
| R7 | R2 | + | R6 | 3 | 4 |

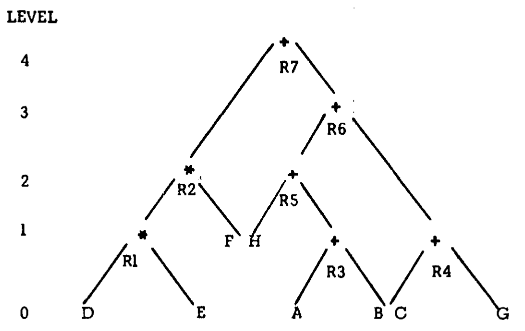


Figure 12—Parallel computation of A+B+C+D*E*F+G+H using Squire's algorithm

## Baer and Bovet's algorithm

The algorithm uses multiple passes. To each pass corresponds a level. All temporary results which can be generated at that level are constructed and inserted appropriately in the output string produced by the corresponding pass. Then, this output string becomes the input string for the next level until the whole expression has been compiled. Thus the number of passes will be equal to the number of levels in the syntactic tree. During a pass the scanning proceeds from left to right and each operator and operand is scanned only once.

The simple intermediate language which this algorithm produces is the most appropriate for multiprocessor compilation in that it shows directly all operations which can be performed in parallel, namely those having the same level number. The syntactic tree generated by this algorithm is shown in Figure 13.

## A new algorithm

This section will introduce a technique whose goals are: (1) to produce a binary tree which illustrates the parallelism inherent in an arithmetic expression; and

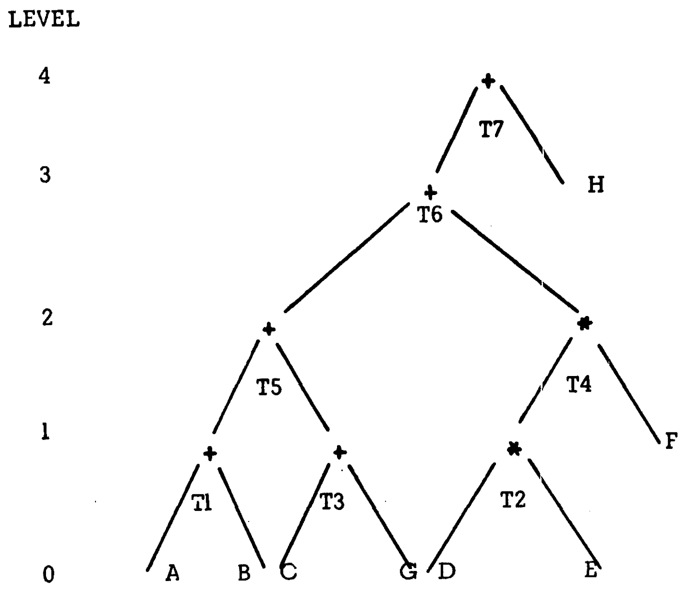


Figure 13—Parallel computation of A+B+C+D*E*F+G+H using Baer and Bovet's algorithm

(2) to determine the number of registers needed to evaluate large arithmetic or Boolean expressions without intermediate transfers to main memory.

This technique is prompted by the fact that existing computing systems possess multiple arithmetic units which can contain a large number of active storages (registers). In addition, the superior memory bandwidths of the next generation of computers will simplify some of the requirements of this technique.

In the material presented below, a complex arithmetic expression is examined to determine its maximum computational parallelism. This is accomplished by repeated rearrangement of the given expression. During this process the given expression in reverse Polish form is also tested for "well formation", i.e., errors and oversights in the syntax, etc.

The arithmetic expression which was used as a model earlier will also be used here, namely A+B+C+D*E*F+G+H. The details of the algorithm follow:

(1) The first step is to rewrite the expression in reverse Polish form and to reverse its order.

$$+H+G+*F*E\ D+C+B+A$$

(2) Starting with the rightmost symbol of the string, assign a weight to each member of the string based on the following procedure:

Assign to symbol $S_i$ the value $V_i = (V_{i-1}) + R_i$ $i = 1,2,\ldots,n$

where $R_i = 1 - \delta(S_i)$ given that

$\delta(S_i) = 0$ if $S_i$ is a variable

$\delta(S_i) = 1$ if $S_i$ is a unary operator

$\delta(S_i) = 2$ if $S_i$ is a binaryoperator

and $V_{i-1} = V_{i-2}+R_{i-1}$, $V_{i-2} = V_{i-3} + R_{i-2}$, etc.,

such that $V_{i-(i-1)} = V_1 = R_1$, and $V_0 = 0$

Using this procedure, the following expression results:

|  | Root Node |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| i | $\boxed{15}$ | 14 | 13 | 12 | \| | 11 | 10 |
| $S_i$ | $\boxed{+}$ | H | + | G | \| | + | * |
| $V_i$ | $\boxed{1}$ | 2 | 1 | 2 | \| | 1 | 2 |

|  |  | $V_m$ |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
| 9 | 8 | $\boxed{7}$ | 6 | 5 | 4 | 3 | 2 | 1 |
| F | * | $\boxed{E}$ | D | + | C | + | B | A |
| 3 | 2 | $\boxed{3}$ | 2 | 1 | 2 | 1 | 2 | 1 |

Note that for a "well-formed expression" of n symbols $V_n = 1$.

(3) At this point the root node of the proposed binary tree can be determined. Thus the given string can be divided into two independent sub-strings. To determine the root node, draw a line to the left of the first symbol with a weight of 1 ($i = 11$, $S_i = +$, $V_i = 1$) to the left of the symbol with the highest weight, $V_m$($i=7$, $S_i=E$, $V_i=V_m=3$). The two independent substrings consist of the strings to the left and to the right of this line. The root node will be the leftmost member of the string to the left of the line ($i=15$, $S_i=+$, $V_i=1$). Note that $V_i$ also equals 3 for $i=9$; however $V_m$ is chosen from the earliest occurrence of a symbol with the highest weight.

(4) The next step is to look for parallelism withni each of the new substrings. Consider the rightmost substring. Form a new substring consisting of the symbols within the values of $V_i = 1$ to the right and to the left of $V_m$. Transpose this substring with the substring to the right of it whose leftmost member has a weight of $V_i = 1$.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| INITIAL RIGHTMOST | $S_i$ | + | * | F | * | E | D | + | C | + | B | A |
| SUBSTRING | $V_i$ | 1 | 2 | 3 | 2 | 3 | 2 | 1 | 2 | 1 | 2 | 1 |

$\longrightarrow \quad \longleftarrow$

|  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FINAL RIGHTMOST | i | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| SUBSTRING | $S_i$ | + | + | C | + | B | A | * | F | * | E | D |
|  | $V_i$ | 1 | 2 | 3 | 1 | 3 | 2 | 1 | 2 | 1 | 2 | 1 |

This procedure is repeated until the initial $V_m$ occupies the position $i=2$ in the substring. For this example this is already the case. Thus the rightmost substring is in the proper form.

(5) The transposition procedure of step 4 is applied next to the leftmost substring. However, since the leftmost substring of this example consists of only two operands and one operator, no further operations are necessary.

(6) The resultant binary tree is shown in Figure 14. The numbers assigned to each node represent the final weight $V_i$ of the symbol as determined in steps 1–5 above.

Some observations and comments on this algorithm are given below.

(1) The two branches on either side of the root node can be executed in parallel. Within each main branch, the transposition procedure of step 4 yields supplementary root nodes. The sub-branches on each side of the supplementary nodes can be executed in parallel.

(2) The number of levels in the binary tree can be

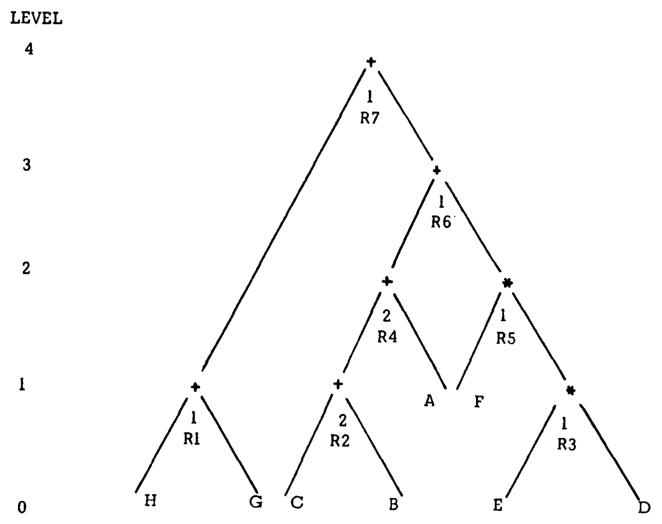


Figure 14—Binary tree for parallel computation of A+B+C+D*E*F+G+H

predicted from the Polish form of the original string.

No. of LEVELS = MAX [NUMBER OF 1's; Vm] in the substring (rightmost or leftmost) containing Vm.

(3) The tree is traversed in a modified postorder form.[20] The resulting expression is

D*E*F+A+B+C+G+H

(4) An added feature of this technique is that the number of registers required to evaluate this expression without intermediate STORE and FETCH operations is obtained directly from the binary tree. This information is provided by the highest weight assigned to any node within the tree. Thus for this example the expression could be evaluated using at most two registers without resorting to intermediate stores and fetches.

(5) This technique of recognizing parallelism on a local level has been applied to a single instruction, in particular, an arithmetic expression. It is worthwhile mentioning that each variable within the expression can itself be the result of a processable task. Thus this technique can be extended to a higher level of parallel stream recognition, i.e., level parallelism.

In order to implement the techniques mentioned here for components within tasks and the techniques mentioned earlier for individual tasks, several system features are desirable. Schemes for detecting parallel processable components within compound tasks are oriented primarily toward arithmetic expressions. For these situations string manipulation ability would be highly desirable. Since individual tasks are represented by a graph and its matrix, the ability to manipulate rows and columns easily would be very important. In this same area, an associative memory could greatly reduce execution time in the implementation of precedence partitions.

## ACKNOWLEDGMENTS

The authors would like to thank the referees of the FJCC for their comments and suggestions which resulted in improvements of this paper.

## REFERENCES

1 A J BERNSTEIN
*Analysis of programs for parallel processing*
IEEE Trans on EC Vol 15 No 5 757-763 Oct 1966

2 E W DJKSTRA
*Solution of a problem in concurrent programming control*
Comm ACM Vol 8 No 9 569 Sept 1965

3 D KNUTH
*Additional comments on a problem in concurrent programming control*
Comm ACM Vol 9 No 5 321-322 May 1966

4 E G COFFMAN R R MUNTZ
*Models of pure time sharing disciplines for research allocation*
Proc 1969 Natl ACM Conf

5 M E CONWAY
*A multiprocessor system design*
Proc FJCC Vol 23 139-146 1963

6 A OPLER
*Procedure-oriented statements to facilitate parallel processing*
Comm ACM Vol 8 No 5 306-307 May 1965

7 J A GOSDEN
*Explicit parallel processing description and control in programs for multi- and uni-processor computers*
Proc FJCC Vol 29 651-660 1966

8 N E ABEL P P BUDNIK D J KUCK
Y MURAOKA R S NORTHCOTE
R B WILHELMSON
*TRANQUIL: A language for an array processing computer*
Proc SJCC 57-68 1969

9 D A FISHER
*Program analysis for multiprocessing*
Burroughs Corp May 1967

10 C V RAMAMOORTHY
*Analysis of graphs by connectivity considerations*
Journal ACM Vol 13 No 2 211-222 April 1966

11 C V RAMAMOORTHY M J GONZALEZ
*Recognition and representation of parallel processable streams in computer programs—II (task/process parallelism)*
1969 Natl ACM Conf

12 C V RAMAMOORTHY
*A structural theory of machine diagnosis*
Proc SJCC 743-756 1967

13 M J GONZALEZ C V RAMAMOORTHY
*Recognition and representation of parallel processable streams in computer programs*
Symposia on Parallel Processor Systems Technologies and Applications Ed. L C Hobbs Spartan Books June 1969

14 E C RUSSELL G ESTRIN
*Measurement based automatic analysis of FORTRAN programs*
Proc SJCC 1969

15 J B DENNIS
*Programming generality, parallelism and computer architecture*
Proc IFIPS Congress 68 C1-C7

16 H HELLERMAN
*Parallel processing of algebraic expressions*
IEEE Trans on E C Vol 15 No 1 Feb 1966

17 H S STONE
*One-pass compilation of arithmetic expressions for a parallel processor*
Comm ACM Vol 10 No 4 220-223 April 1967

18 J S SQUIRE
*A translation algorithm for a multiprocessor computer*
Proc 18th ACM Natl Conf 1963

19 J L BAER D P BOVET
*Compilation of arithmetic expressions for parallel computation*

Proc IFIPS 68 B4-B10

20 D KNUTH
*The art of computer programming, Vol. 1, fundamental algorithms*
Addison-Wesley 316

21 R S NORTHCOTE
*Software developments for the array computer ILLIAC IV,*
Univ of Illinois Rpt No 313 March 1969

# Performance modeling and empirical measurements in a system designed for batch and time-sharing users

*by* JACK E. SHEMER and DOUGLAS W. HEYING

*Scientific Data Systems, a Zerox Company*
El Segundo, California

## INTRODUCTION

If any design goal is common to all computer system organization schemes, it is that of providing "effective service" both *externally* to the user of the computational facility and *internally* with respect to utilization of system resources. Thus, generally speaking, there are at least two dimensions to this design objective. On the one hand, effective service is the external satisfaction of a broad spectrum of user demands. For example, the ideal system might be visualized as one which economically provides a large number of programming languages; machine compatibility with other computers of widely diverse hardware; and rapid computation. On the other hand, effective service is the internal utilization of all system components so as to increase computational efficiency. In this respect, system structures are implemented which strive to maximize sub-system simultaneity and system throughput. For example, a degree of macro-parallelism is attained in many present day systems by allowing a central processing unit (CPU) and input/output controller to share the use of a main memory register, thereby enabling processing and input/output (I/O) to proceed concurrently (for one or several independent programs, depending upon the system software).

In general, external effectiveness is all that the user sees, and it is therefore of primary interest to him. Whereas, the purveyor of the equipment is vitally concerned with internal utility and coordination. However, this latter consideration indirectly relates to the quality of service the user receives (his waiting time for service completion, the price he is charged for service, etc.).

The ramifications of hardware and software designs to achieve such service can be investigated both internally and externally; yet, a particular design strategy need not supplement effective service from both viewpoints. On the contrary, schemes tailored to improve external utilization often degrade internal service effectiveness and vice versa. Unfortunately, in confronting these design trade-offs, the designer often had to rely upon heuristic and intuitive arguments, since there is a general lack of design models which quantitatively relate system variables to reflect a priori performance estimates. Hence, the design is complicated not only by trade-offs between the often dissimilar aims of external and internal effective service, but also by a deficiency of design tools for investigating various implementation alternatives.

These problems are especially amplified with the advent of time-shared computer systems. In time-sharing systems, an ideal goal is to respond to interactive on-line users such that each user receives the impression that he has his own computer, yet at a price he can afford. Thus in these systems, the computer complex is shared among a number of independent users who are concurrently communicating with the system, generating programs and interactive service requests via on-line remote terminal equipment. This action enables one to achieve economies of scale and distribute the cost of the system among all users according to their usage of the facilities. Similarly, the objective of rapid response is realized by time slicing CPU service and sharing it among the on-line users. A request for program execution is not necessarily serviced to completion; but rather jobs are granted finite intervals (quanta) of processing time. If a job fails to exhaust its demands during a quantum allocation, then it is truncated and postponed according to a scheduling discipline, thereby facilitating rapid response to short requests.[1–4] This preferential treatment of short jobs increases the programmer's productiveness, since one-attempt efforts, editing, debugging, and other typically short interactive demands often encounter exorbitant turn-around times in batch processing environments (i.e., in relation to the amount of actual processing time consumed, due to problems of key punching, printer output, card stacking, and total system demand).

However, since computation is not necessarily run to completion and main memory size is limited (by both economic and physical reasons), programs must be swapped into and out of main memory as the CPU commutates its service from request to request. Therefore, unless swapping is achieved with no loss in time, it is obvious that service in the time-sharing sense is less efficient in CPU utilization than service to completion. Also, the time spent scheduling, allocating buffers, and controlling swap input/output represents overhead or wasted processing time which, due to incomplete servicing, is greater in time-sharing systems than batch processing systems. Furthermore, if the system is dedicated to servicing on-line requests, the CPU is essentially idle during periods of low on-line input traffic. Hence, a design compromise must be attained between external response rapidity and internal efficiency since system performance, in the general case, is a function of both response to selected classes of users and utilization of system resources.

Yet, exploring such problem areas prior to design is complicated, because any performance investigation is incorrigibly statistical. Performance is not only a function of software characteristics such as the input/output, memory, and processing requirements of each on-line request together with the occurrence rate of such requests, but also dependent upon hardware characteristics such as the instruction processing rate and rates accessing secondary memory.

This paper presents one approach to mitigating some of these difficulties. A system design is briefly described and then analyzed utilizing a mathematical model. The system is structured to accommodate both batch and time-sharing users with the goal being to achieve a balance of system efficiency and responsiveness. A set of variables are defined which characterize on-line user demands and the servicing capacity of various units within the system. These variables are then quantitatively related in a mathematical model to derive salient performance measures. Examples are given which graphically display these measures versus various ranges of the system variables. These a priori performance estimates are then compared with empirical data extracted from the system during its actual operation. Here the emphasis is given to mathematical modeling because this analysis method is more expedient and generally less costly than the alternative approach of simulation. Moreover, since many of the variables are non-independent and rely upon characterization of user demands, and since these are difficult to accurately describe prior to actual operation, the macroscopic and statistical indications provided by a mathematical model are perhaps all that one can feasibly obtain.

*Design and performance study*

## System design

The Batch/Time-Sharing Monitor (BTM) is designed to afford SDS Sigma 5 and Sigma 7 users with interactive and on-line time-sharing without disrupting batch operations. For considerations of efficiency, the primary objective of the BTM design is to provide limited time-sharing service while concentrating on throughput of batch jobs—the servicing of time-sharing users is allocated to minimize response for interactive users with no special service given to the compute bound on-line users (because high-efficiency batch service is available).

Thus, the system is structured with resources for the batch and time-sharing portions of the system separated as much as possible. Different areas of main memory are allocated so that a (compute bound) batch user is always "ready to run." The file device is common because files may be shared between batch and time-sharing users. However, the management technique used minimizes the interference from this factor. The swapping Rapid Access Disc (RAD) for time-sharing users is independent of the file device, thus insuring that swaps in process do not affect on-going batch programs.

The batch user is kept essentially compute bound by buffering all of his unit record I/O via a RAD. This allows the compute portion of each job to follow that of the previous job without waiting for the printout, etc., to complete. Thus, there is no need to attempt to reclaim swap time from one time-sharing user to another—a natural claimant: the batch job is readily available.

Hence, a very simple (and low overhead) swapping and scheduling algorithm can be used. As a particular user is dismissed, other users are polled in turn to see who is "ready to run." If someone is found (not the same user), a replacement swap is initiated and the CPU is allocated to the batch job. When the swap-out/swap-in is complete, the new user is given one quantum (i.e., providing the batch job has already had at least its quantum); then the cycle is repeated.

In this way, batch is guaranteed a certain percentage of the machine (and typically gets much more), and a moderate number of time-sharing users receive rapid response to conversational request. Yet with this relatively simple framework, a number of questions are unavoidable: How does on-line response and batch throughput vary with the number of on-line users, and how do other variables such as quantum size and swap time relate to system performance? Moreover, how does one characterize system performance and the variables which influence it?

## Parameterizations and performance measures

The subject of "on-line" response is unfortunately plagued by many interpretations of what constitutes response (and, moreover, what defines adequate response). For the purposes of this paper, "typical on-line requests" are those which require minimal central processor time—less than one quantum allocation. Thus, the response time $C_1$ to a "typical on-line demand" is that period elapsing between request generation (the keying in of a control character such as "carriage return") and the termination of the *first* time quantum* which is allocated to the servicing of the request. This definition provides the basis upon which the on-line performance of the BTM system is analyzed in this paper, since it is assumed that on-line users are typically in phases of program preparation.** Thus, providing the quantum is large enough, the great majority of user interactions (e.g., "open the next line," "delete source image," "perform syntax check and insert into text," etc.) can be satisfied with single quantum allocations.

The mathematical model developed in the Appendix enables one to characterize the system by selecting values for the variables:

$N$ = total number of active on-line communication sources (i.e., the number of remote users who are *concurrently* using the system).

$\lambda$ = average user interaction rate (frequency at which a single user requests service by the CPU).

$\mu$ = mean rate at which on-line requests are serviced by the CPU ($1/\mu$ = average amount of CPU time required to complete each request given that the CPU was dedicated to the servicing of the request).

$\bar{S}$ = the average amount of time required to swap an old user out of core and load a new user (clearly, $\bar{S}$ is dependent upon the swapping device as well as program size).

$q_R$ = time quantum allocated to on-line requests (time-sharing users).

$q_B$ = time quantum given to batch requests (background users).

$\bar{m}$ = the average cumulative quantum extension (for monitor services such as scheduling, file I/O, service calls, etc.) incurred during the period elapsing between successive quantum allocations to on-line jobs.

To supplement analysis efforts, the BTM system software is capable of monitoring these (and other) variables and accumulating their statistical distributions during actual system operation. This does not impose any significant overhead since much of this data is already accumulated in the accounting log, and (as in many other commercial systems) used as a basis for charging users.

Upon establishing reasonable values for the above variables, the model can then be used to derive performance measures. In terms of response, the salient performance index is $E[C_1]$ where

$E[C_1]$ = the expected response time which "typical on-line demands" experience (see definition given above).

In addition, the model can readily be used to estimate the percentage of CPU time available for batch jobs; the percentage of CPU time received by time-sharing users; utilization of the swapping RAD; expectations of system revenues; and a variety of other indices obtained from combinations of the derived parameters.

A priori estimates for some of these performance measures are given in Figures 1-5 for reasonable ranges

* Also note that if the scheduling algorithm is round-robin then $C_1$ provides a basis for approximating the response time for a request which requires multiple quanta.[4]

** Note that this is not the case in system environments in which the on-line users run production (compute bound) programs.

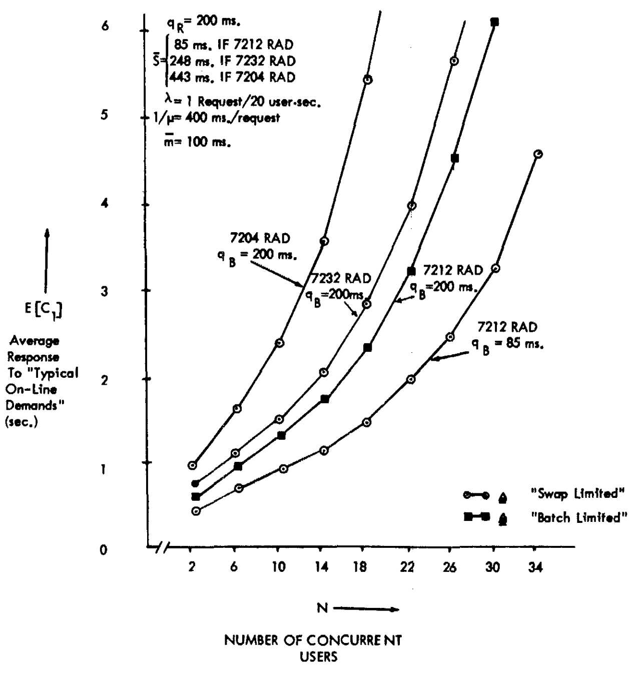


Figure 1—E[C1] vs. N ($\mu$ = 2.5 requests/sec.)

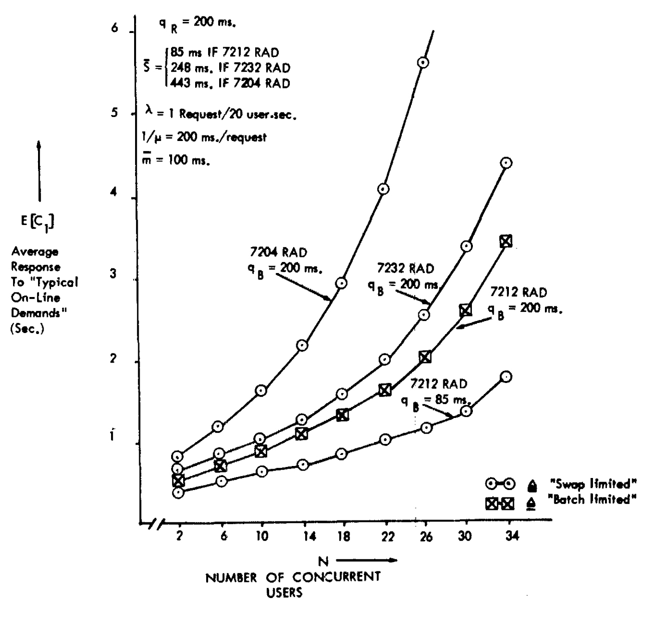


Figure 2—E[C1] vs. N ($\mu$ = 5 requests/sec.)

of the variables N, $\lambda$, $\mu$, $\bar{S}$, $q_R$, $q_B$, and $\bar{m}$. Obviously, these variables will differ from one environment to another. Therefore, before discussing conclusions which can be drawn from these graphical results, it is appropriate to clarify the parameterizations and assumptions which were used in the calculations:

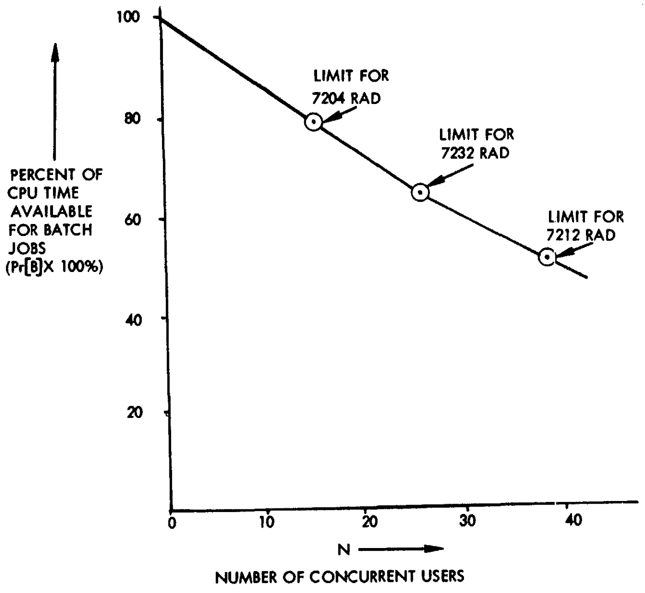


Figure 3—Relative batch capability

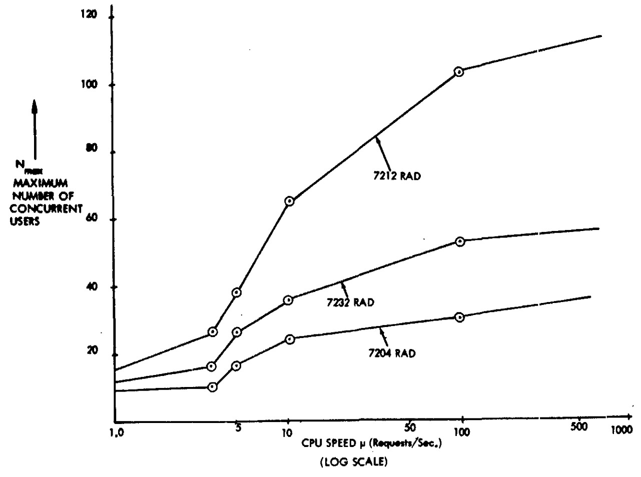


Figure 4—$N_{max}$ vs. CPU speed $\mu$

1. The average swap time $\bar{S}$ was conservatively calculated assuming that four RAD accesses are required per swap with an average total of 16K words transferred during each swap. (The RAD's are head per track rotating memories operating at 1800 rpm; and the SDS model 7204, 7232 and 7212 RAD transfer data at rates 187 $\times$ $10^3$ bytes/sec., 384 $\times$ $10^3$ bytes/sec. and 3 $\times$ $10^6$ bytes/sec., respectively.)
2. The user interaction rate $\lambda$ was estimated from statistics gathered at RAND[5] and other data extracted from the GE/Dartmouth BASIC system[6] and the SDS 940 system.

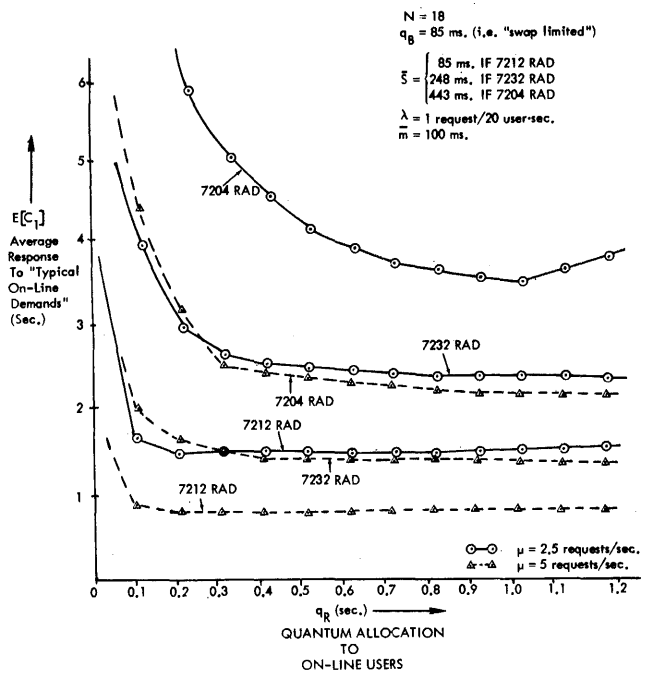


Figure 5—E[$C_1$] vs. $q_R$ (N = 18)

3. The selection of $q_R$ = 200 ms. was established such that the majority of user interactions are satisfied with single quantum allocations. Whereas, selecting $q_B$ = 85 ms. and 200 ms. was done merely to demonstrate "swap limited" and "batch limited" operation, respectively.
4. The value of the average monitor time $\bar{m}$ per on-line/batch quantum cycle was approximated utilizing batch accounting information and timing studies of monitor services.
5. Values of $\mu$ were chosen such that the average on-line quantum $\bar{q}_R$ would be $\approx$ 125 ms. to 150 ms. when 200 ms. was allocated. This selection was inferred from data extracted from the SDS 940 System and BTM code traces. (Yet, note that a single parameter $\mu$ does not provide a characterization covering the more general case in which the processing time distribution is multi-modal.† However, for purposes of studying interactive response, it provides a good approximation and lends itself to the mathematical analysis.)

† The multi-modal case arises because of a multiplicity of language facilities and the natural division of requests into interactive or compute demands.

## Mathematical results

Given this framework, let us now turn our attention to the figures. Employing the mathematical model, a priori estimates of average interactive response time E[$C_1$] are displayed versus N in Figure 1 and Figure 2 for $\mu$ = 2.5 requests/sec. and $\mu$ = 5 requests/sec., respectively. Here, three different curves are plotted in each figure to demonstrate the limiting effects of each swapping device (i.e., "swap limited" operation when the batch quantum $q_B$ is less* than the swap time S). Also, note that an additional curve is given for the model 7212 RAD to display the effects of selecting a batch quantum which exceeds the swap time (i.e., "batch limited" operation). This latter curve shows that the fastest swapping device effectively becomes a slower device when $q_B$ is set such that operation is "batch limited"—the model 7212 RAD is almost equivalent to a model 7232 RAD when $q_B$ = 200 ms.

Now since N is the total number of concurrent users (active communication sources), Figures 1 and 2 enable one to estimate a value for the maximum number of users $N_{max}$ which the system can simultaneously accommodate by: (1) assuming "swap limited" operation and (2) defining what constitutes adequate response to typical on-line demands. For example, if one assumes that adequate interactive response is achieved if $\approx$ 80% of the time a user experiences a delay of less than 5 sec. then, depending upon $\mu$, one concludes:**

i. the model 7204 RAD will accommodate a maximum of 10 to 16 concurrent users for*** $\mu$ = 2.5 requests/sec. to $\mu$ = 5 requests/sec., respectively;
ii. the model 7232 RAD will accommodate a maximum of 16 to 26 concurrent users for $\mu$ = 2.5 requests/sec. to $\mu$ = 5 requests/sec., respectively;
iii. the model 7212 RAD will accommodate a maximum of 26 to 38 users for $\mu$ = 2.5 requests/sec. to $\mu$ = 5 requests/sec., respectively.

However, the actual number of on-line users who

* For this situation, the actual batch quantum allocation is the swap time S.

** These conclusions were made by assuming that the probability distribution for response time $C_1$ is such that twice the mean E[$C_1$] is (at least) the 80 percent point. This is a reasonable assumption in light of both the mathematical characterizations used in the model and empirical measuresments.

***Note that reducing $\mu$ from 5 requests/sec. to 2.5 requests/sec. is tantamount to reducing processing speed by a factor of 1/2.

concurrently use the system is a statistical parameter which generally is less than $N_{max}$ and varies according to the total number of on-line subscribers, their demands, processing speed, $N_{max}$, etc. In practice, the total number of on-line subscribers typically exceeds $N_{max}$ by at least a factor of three.

For the above cases, nominally 50–80% of the CPU time is available for batch jobs. This is shown in Figure 3. Similarly, utilizing this same response criterion, it is interesting to observe the effects of increasing**** CPU speed $\mu$. This is demonstrated in Figure 4 for each of the swapping devices. As CPU speed increases indefinitely, the capacity of the system to service on-line requests approaches a limit established by the swapping device.

Additional insight into system responsiveness is provided by Figure 5. Here, $E[C_1]$ is graphically displayed versus the on-line user quantum $q_R$ for "swap limited" operation and $N = 18$ (with all other variables the same as those employed in Figures 1 and 2.) Note that the selection of a minimum $q_R$ is very critical; however, having established a minimum $q_R$, the variations are not dramatic for a relatively large range above minimum $q_R$. Also, notice that as $\mu$ is reduced from 5 requests/sec. to 2.5 requests/sec., a model 7232 RAD must be used to achieve what a model 7204 RAD accomplished in the former case; and similarly, a model 7212 RAD is required to equal the performance of a model 7232 RAD.

## Experimental results

Extensive statistics were gathered from the system (while running typical jobs) with a twofold purpose in mind. First, it was necessary to substantiate the validity of the assumptions employed in the model; i.e., establish that the chosen parameters were indeed consistent with the actual environment. Secondly, a correlation between empirically measured performance and the results of the model would lend credence to the validity of the model, and therefore allow us to extrapolate and predict performance for other user environments and system configurations.

The first objective was accomplished by observing a BTM system which used a model 7212 RAD for swapping with quanta $q_R = q_B = 200$ ms. Values for $\lambda$, $\mu$, $\overline{m}$ and program size were tabulated for many different observation periods. For each of these monitoring sessions different average values were obtained, but the values $\mu = 3.5$ requests/sec., $\lambda = 1$ request/15 user-sec., $\overline{S} = 85$ msec. and $\overline{m} = 100$ msec. were found to be quite representative of most samples. The variables $\mu$ and $\lambda$ were most subject to variation and ranged from 2 to 6 requests/sec. and from 1 request/25 user·sec. to 1 request/10 user·sec., respectively. Also, the data indicated that the assumptions of exponentially distributed CPU time and request inter-arrival time provided good approximations of user demands.

Given that the first objective was satisfied, realization of the second objective is buttressed by Figure 6 which plots the average of all sampled values for two of the key performance indications (average response time $E[C_1]$ and CPU time available for batch Pr[B]) as a function of the number of users N. Upon comparing these results with the mathematical predictions (also see Figures 1–3), one can infer that (at least for the range of variables considered) the mathematical model is reasonably consistent with actual system operation.

### *Comments*

The analysis presented above primarily focused attention on the system's capacity to accommodate user demands. Even though no mention was given to cost/performance tradeoffs, the model lends itself to this latter design consideration. For example, the variables N, Pr[B], and $\mu$ might be combined to reflect the revenue derived for service to batch jobs and the revenue obtained for servicing interactive users which could then be weighted against the cost expended to

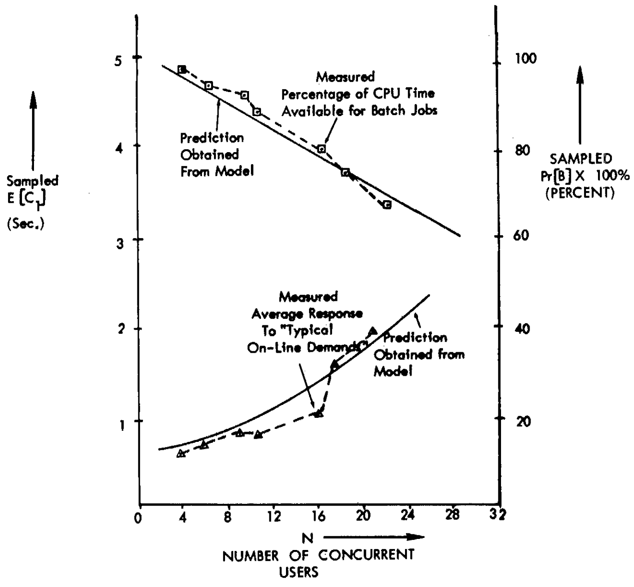


Figure 6—Empirical results

**** Note that this latitude is only possible on a limited basis (e.g., code optimization, faster memory, faster operation unit, multi-processing, etc.)

provide (and maintain) the system complement. This would provide a basis for the designer to balance CPU cost/performance with that of other system elements.

The process of selecting and examining performance indexes similar to those discussed here enables the designer to better appraise the many implementation tradeoffs which confront him. Moreover, when supplemented with empirical data, these techniques provide a basis for not only configuring existing systems but also synthesizing new systems. However, it should be emphasized that apart from the mathematical model itself and its macroscopic treatment of the system, the fidelity of the results and conclusions obtained in this analysis (or any analysis of this sort) can only be as good as the accuracy attributed to the independent variables (N, λ, μ, m, S). The values possessed by these variables dramatically affect performance and will vary from one environment to another. Therefore, one should be cautious before inferring any explicit and universal characterizations of system performance.

## REFERENCES

1 B KRISHNAMOORTHI R C WOOD
*Time-shared computer operations with both interarrival and service time exponential*
J A C M Vol 13 317–338 July 1966
2 E G COFFMAN JR
*Stochastic models of multiple and time-shared computer operations*
Report 66–38 Dept of Eng Univ of Calif Los Angeles June 1966
3 L KLEINROCK
*Time-shared systems: A theoretical treatment*
J A C M Vol 14 242–261 April 1967
4 J E SHEMER
*Some mathematical considerations of time-sharing scheduling algorithms*
J A C M Vol 14 262–272 April 1967
5 G E BRYAN
*JOSS: 20,000 hours at a console—a statistical summary*
Proc F J C C 769–777 1967
6 H CANTRELL
*Time-sharing data*
General Electric Technical Information Series Report R65CD12 December 1965
7 T L SAATY
*Elements of queueing theory*
McGraw-Hill New York 1961

## APPENDIX

*BTM mathematical model*

Consider the generation of on-line requests on each communication channel is an exponential process with parameter λ. Hence, the time interval x between completion of a request and generation of a new request on a given line is described by the distribution function

$$A(x) = \begin{cases} 1 - e^{-\lambda x} & \text{for } x \geq 0 \\ 0 & \text{for } x < 0 \end{cases}$$

Similarly, assume that the service time t required by each on-line request is exponentially distributed with parameter $\mu$ and characterized by the distribution function

$$B(t) = \begin{cases} 1 - e^{-\mu t} & \text{for } t \geq 0 \\ 0 & \text{for } t < 0 \end{cases}$$

Given that there are N channels, let $\rho$ (t) denote the probability that n on-line requests are queued at some arbitrary time t for n = 0, 1, ···N, then

$$\frac{d\rho_n(t)}{dt} = \begin{cases} -N\lambda\rho_0(t) + \mu \Pr[R(t)]\rho_1(t) \\ \qquad \text{for } n = 0 \\ [(N-n)\lambda + \mu\Pr[R(t)]]\rho_n(t) \\ \quad + (N-n+1)\lambda\rho_{n-1}(t) \\ \quad + \mu\Pr[R(t)]\rho_{n+1}(t) \\ \qquad \text{for } 0 < n < N \\ \mu\Pr[R(t)]\rho_N(t) + \lambda\rho_{N-1}(t) \\ \qquad \text{for } n = N \end{cases}$$

where Pr[R(t)] denotes the probability that at time t the computer is servicing one of the remotely generated on-line requests. Note that in the above equations, the input rate is (N − n)λ when n requests are queued. Thus the model accounts for the natural variations in demand intensity which result because there are a finite number N of input sources.

From these equations, the stationary probability[7] that n on-line requests are queued is

$$\rho_n = \frac{N!}{(N-n)!}\left(\frac{\lambda}{\mu \Pr[R]}\right)^n \rho_0$$

where

$$\Pr[R] = \lim_{t \to \infty} \Pr[R(t)] \text{ and}$$

$$\rho_0 = \frac{1}{\left[1 + \sum_{n=1}^{N} \frac{N!}{(N-n)!}\left(\frac{\lambda}{\mu \Pr[R]}\right)^n\right]}$$

The probability Pr[R] can be estimated by considering the interval which elapses between successive allocations of a quantum to on-line users. Let $T_k$ denote the total time between the 0th on-line quantum completion and the kth on-line quantum completion. If the kth completion leaves the on-line queue in an empty state, then the expected value of the time $\Delta T_k$ until the next on-line quantum completion is

$$E[\Delta T_{k|n=0}] = \frac{1}{N\lambda} + \bar{q}_B + \bar{q}_R + \bar{m}$$

where $\bar{q}_B$ is the average quantum which batch users receive; $\bar{q}_R$ is the expected duration of an on-line (remote user) quantum; $(1/N\lambda)$ is the mean time until the generation of the next on-line request; and $\bar{m}$ is the expected monitor overhead time per batch/on-line quantum cycle. Here, $\bar{m}$ accounts for any scheduling; I/O overhead; file operations, and any other CPU time pre-empted by the monitor which results during the cycle of a quantum allocation to a batch job followed by a quantum allocation to an on-line job.

In the case when the kth on-line quantum completion does not leave the interactive user queue empty, then with probability $(1 - \rho_0)$

$$E[\Delta T_{k|n\geq 1}] = \bar{q}_B + \bar{q}_R + \bar{m}$$

Now let $T_B$, $T_R$, and $T_m$ denote respectively the length of time out of $T_k$ which the system spends servicing batch jobs, on-line jobs, and monitor functions, respectively.

Then as k goes to infinity, the ratios $T_B/k$, $T_R/k$, and $T_m/k$ converge with probability one to $(\bar{q}_B + \rho_0/N\lambda)$, $\bar{q}_R$, and $\bar{m}$, respectively. Therefore, in the limit, an approximation to the fraction of the time which the system spends servicing on-line requests is

$$\Pr[R] = \lim_{k\to\infty}\left[\frac{T_R}{T_k}\right] = \lim_{k\to\infty}\left[\frac{T_R/k}{T_k/k}\right] = \left[\frac{\bar{q}_R}{\bar{q}_R + \bar{q}_B + \bar{m} + \rho_0\left(\frac{1}{N\lambda}\right)}\right]$$

Then, noting that $\bar{q}_B + \bar{q}_R$ is essentially the computation cycle, this leads one to express Pr[R] as

$$\Pr[R] = \frac{\bar{q}_R}{f(\bar{q}_B + \bar{q}_R)} \text{ where } f > \left(1 + \frac{\bar{m}}{\bar{q}_B + \bar{q}_R}\right)$$

Here, f is an appropriate scale factor introduced to facilitate solving for $\left\{\rho_n\right\}_{n=0}^{N}$. The numerical technique is to let f increase by some small $\Delta f$ until a solution for $\rho_o$ is obtained which is consistent with Pr[R]. The variable f satisfying this criterion will vary dramatically depending upon N, $\bar{m}$, $\mu$, $\lambda$ and $\bar{q}_B$.

Upon solving for $\rho_0$, the percentage of CPU time available for batch jobs is

$$\Pr[B] = \frac{\bar{q}_B + \rho_0\,(1/N\lambda)}{\bar{q}_R + \bar{q}_B + \bar{m} + \rho_0\,(1/N\lambda)}$$

The variables $\bar{q}_B$ and $\bar{q}_R$ are heavily influenced by quantum periods and swap time. If one assumes that (with the exception of a batch quantum allocation every other quantum) on-line jobs run on a demand basis (i.e., the batch quantum $q_B$ is less than the swap time S), then $\bar{q}_B = \bar{S}$. Hence, the swap time limits the rate at which successive quantum allocations are provided to the on-line requests (i.e., maximum service capacity is given to on-line requests). Whereas, if the batch quantum limits the servicing of on-line requests ($q_B > S$), then $\bar{q}_B = q_B$. Therefore, for completeness

$$\bar{q}_B = \begin{cases} q_B \text{ if } S < q_B \\ \bar{S} \text{ if } S \geq q_B \end{cases}$$

and from the exponential distribution of service time for on-line requests[1-4]

$$\bar{q}_R = 1/\mu\,(1 - e^{-\mu q_R})$$

Given the foregoing relations, it is now possible to estimate the expected cycle time $E[C_1]$ which an individual on-line (remote user) request experiences before it is granted its first quantum allocation. As emphasized in an earlier section, attention is focused upon $E[C_1]$ since it provides an indication of the responsiveness of the system to handling "typical on-line requests" which require less than one quantum of CPU time. By considering the system only at epochs of transition between batch to on-line, monitor to batch (or on-line), idle to batch (or on-line), etc., one obtains the approximation

$$E[C_1] \approx \rho_0\, E[T_0] + E[n]\,(\bar{q}_B + \bar{q}_R + \bar{m}) + \bar{q}_R$$

where

$$E[n] = \sum_{n=1}^{N} n\rho_n$$

and $E[T_0]$ is the expected time remaining subsequent to the arrival of an on-line request before the next quantum allocation is initiated. The value of $E[T_0]$ is difficult to accurately express since it is a function of the probability densities for $q_B$ and m together with machine state probabilities; however, it is clear that

$$E[T_0] \leq [\bar{q}_B + \bar{m}]$$

At any rate, $E[T_0]$ is not a dominant factor in $E[C_1]$ unless $E[C_1]$ is extremely small (i.e., $E[C_1] \approx q_R + E[T_0]$, for example). Hence, the precise value of $E[T_0]$ is not critical in those cases which are of particular interest (namely, those resulting when the on-line queue tends toward saturation; i.e., $E[n] \rightarrow N$).

In addition to the above result for $E[C_1]$, since the scheduling discipline is round-robin, it is possible to estimate[2–4] the expected total response time $E[r|t]$ for an on-line request which requires a processing time t in excess of a single quantum $q_R$

$$E[R|t] \cong t + \langle t/q_R \rangle \, [E[C_1] - (\rho_0 \, E[T_0] + \bar{q}_R) + \bar{q}_B + \bar{m}]$$

where $\langle a/b \rangle$ is the smallest integer greater than a/b.

*Alternate model*

Let $\rho_{mn}(T_k)$ denote the probability that n on-line requests are queued at epoch $T_k$ marking the completion of the k$^{\text{th}}$ on-line quantum allocation, given that at epoch $T_{k-1}$ there were m on-line requests awaiting service from the system.[1,2] Then independent of k since the CPU servicing of requests is characterized as an exponential process

$$\rho_{mn} = \begin{cases} e^{-\mu q_R} \displaystyle\int_0^y \Pr[n - m|m, q_R + t] p_B(t) \, dt \\ \quad + \displaystyle\int_0^{y+q_R-\epsilon} \Pr[n - m + 1|m, t] \, p_{B+R}(t) \, dt \\ \qquad\qquad \text{for } 1 \leq m \leq n \\ 0 \text{ for } n \leq m - 2;\ m \geq 1 \\ \displaystyle\int_0^{y+q_R-\epsilon} \Pr[0|m, t] \, p_{B+R}(t) \, dt \\ \qquad\qquad \text{for } n = m - 1 \geq 0 \end{cases}$$

where $\epsilon \rightarrow 0$ and $\Pr[k|m,t]$ denotes the conditional probability of generating k new on-line requests in a time interval t given that m requests are queued. For example, with exponential inter-arrival

$$\Pr[k|m, t] = \binom{N - m}{k} (1 - e^{-\lambda t})^k (e^{-\lambda t})^{N-m-k}$$

Also, in the above equations

$$y = \begin{cases} S_{\max} \text{ if service to on-line customers is swap limited (i.e., } q_B < S) \\ q_B \text{ if batch quantum limits on-line service (i.e., } q_B \geq S) \end{cases}$$

Here, $p_B$ denotes the probability density function which describes the batch quantum allocation, and $p_{B+R}$ is the convolution of $p_B$ with the density function $p_R$ defining the distribution of an on-line quantum allocation. Both $p_B$ and $p_R$ include overhead functions to account for file I/O, monitor overhead, etc.

The density function $p_B$ is derived from the swap time distribution when $q_B < S$; whereas, it depicts the CPU servicing of batch requests when $S < q_B$. For example, in the latter case with $\delta(z)$ representing the Dirac-delta function describing an independent variable z, one could characterize the constant batch allocation interval by

$$p_B(t) = \delta(t - (\gamma_B + q_B))$$

where the constant $\gamma_B$ reflects batch overhead. Similarly, letting $\gamma_R$ denote the overhead incurred during an on-line quantum allocation

$$p_R(t) = \begin{cases} 0 \quad \text{for } t \leq \gamma_R \text{ or } t > \gamma_R + q_R \\ \mu e^{-\mu t} + e^{-\mu q_R} \, \delta(t - (q_R + \gamma_R)) \\ \qquad \text{for } \gamma_R \leq t \leq \gamma_R + q_R \end{cases}$$

For completeness, the transitions from the 0-state are assumed to be

$$\rho_{on} = \int_0^y \Pr[n|o, t] \, p_B(t) \, dt$$

Then, having formulated the state transitions $\{\rho_{mn}\}$ and defined the density functions $p_B(t)$ and $p_{B+R}(t)$, the problem remains to solve for the steady-state probabilities. This is accomplished by noting that the $p_{mn}$'s define an ergodic Markovian chain whereby in matrix form with $\underline{P} = (\rho_{mn})$ there exists a unique set of numbers $\{\rho_m\}_{n=0}^{N}$ such that

$$(\rho_0 \rho_1 \cdots \rho_N) \, p = (\rho_0 \rho_1 \cdots \rho_N)$$

and

$$\sum_{n=0}^{N} \rho_n = 1$$

The solution of these equations produces the limiting stationary probabilities $\{\rho_n\}_{n=0}^{N}$ which could be used in calculating E[n] to provide a more accurate estimate of E[$C_1$]. (That is, providing one can accurately describe $p_B$, $p_{B+R}$, $\lambda$, etc.).

However, since the accuracy of such variables would be highly questionable in the absence of any empirical information and since this latter model presents a number of non-trivial mathematical difficulties, it was not utilized to derive the results given in this paper. Yet, in the future, as sufficient data is accumulated from the actual operation of BTM systems, then the latter model will enable us to extrapolate and better predict the effects of alterations to the system (e.g., improvements resulting from faster swapping devices or increases in CPU speed).

## ACKNOWLEDGMENT

The authors are indebted to M. Leavitt, D. Cumming, J. Doeppel, T. Martin and G. E. Bryan for their many contributions to the BTM design effort and also wish to extend thanks to all those other individuals at Scientific Data Systems who helped to make this project possible. In particular, the authors are grateful to D. Cota, E. Maso and Dr. R. Spinrad for their guidance in these efforts.

# Dynamic protection structures

*by* B. W. LAMPSON

*Berkeley Computer Corporation*
Berkeley, California

## INTRODUCTION

A very general problem which pervades the entire field of operating system design is the construction of protection mechanisms. These come in many different forms, ranging from hardware which prevents the execution of input/output instructions by user programs, to password schemes for identifying customers when they log onto a time-sharing system. This paper deals with one aspect of the subject, which might be called the meta-theory of protection systems: how can the information which specifies protection and authorizes access, itself be protected and manipulated. Thus, for example, a memory protection system decides whether a program P is allowed to store into location T. We are concerned with how P obtains this permission and how he passes it on to other programs.

In order to lend immediacy to the discussion, it will be helpful to have some examples. To provide some background for the examples, we imagine a computation C running on a general multi-access system M. The computation responds to inputs from a terminal or a card reader. Some of these look like commands: to compile file A, load B and print the output double-spaced. Others may be program statements or data. As C goes about its business, it executes a large number of different programs and requires at various times a large number of different kinds of access to the resources of the system and to the various objects which exist in it. It is necessary to have some way of knowing at each instant what privileges the computation has, and of establishing and changing these privileges in a flexible way. We will establish a fairly general conceptual framework for this situation, and consider the details of implementation in a specific system.

Part of this framework is common to most modern operating systems; we will summarize it briefly. A program running on the system M exists in an environment created by M, just as does a program running in supervisor state on a machine unequipped with software. In the latter case the environment is simply the available memory and the available complement of machine instructions and input/output commands; since these appear in just the form provided by the hardware designers, we call this environment the *bare machine*. By contrast, the environment created by M for a program is called a *virtual* or *user machine*.[6] It normally has less memory, differently organized, and an instruction set in which the input/output at least has been greatly changed. Besides the machine registers and memory, a user machine provides a set of *objects* which can be manipulated by the program. The instructions for manipulating objects are probably implemented in software, but this is of no concern to the user machine program, which is generally not able to tell how a given feature is implemented.

The basic object which executes programs is called a task or *process*;[5] it corresponds to one copy of the user machine. What we are primarily concerned with in this paper is the management of the objects which a process has access to: how are they identified, passed around, created, destroyed, used and shared.

Beyond this point, three ideas are fundamental to the framework being developed:

1. Objects are named by *capabilities*,[3] which are names that are protected by the system in the

sense that programs can move them around but not change them or create them in an arbitrary way. As a consequence, possession of a capability can be taken as prima facie proof of the right to access the object it names.

2. A new kind of object called a *domain* is used to group capabilities. At any time a process is executing in some domain and hence can exercise the capabilities which belong to the domain. When control passes from one domain to another (in a suitably restricted fashion) the capabilities of the process will change.
3. Capabilities are usually obtained by presenting domains which possess them with suitable authorization, in the form of a special kind of capability called an *access key*. Since a domain can possess capabilities, including access keys, it can carry its own identification.

A key property of this framework is that it does not distinguish any particular part of the computation. In other words, a program running in one domain can execute, expand the computation, access files and in general exercise its capabilities without regard to who created it or how far down in any hierarchy it is. Thus, for example, a user program running under a debugging system is quite free to create another incarnation of the debugging system underneath him, which may in turn create another user program which is not aware in any way of its position in the scheme of things. In particular, it is possible to reset things to a standard state in one domain without disrupting higher ones.

The reason for placing so much weight on this property is two-fold. First of all, it provides a guarantee that programs can be glued together to make larger programs without elaborate pre-arrangements about the nature of the common environment. Large systems with active user communities quickly build up sizable collections of valuable routines. The large ones in the collections, such as compilers, often prove useful as sub-routines of other programs. Thus, to implement language X it may be convenient to translate it into language Y, for which a compiler already exists. The X implementor is probably unaware that Y's implementation involves a further call on an assembler. If the basic system organization does not allow an arbitrarily complex structure to be built up from any point, this kind of operation will not be feasible.

The second reason for concern about extendibility is that it allows deficiencies in the design of the system to be made up without changes in the basic system itself, simply by interposing another layer between the basic system and the user. This is especially important when we realize that different people may have different ideas about the nature of a deficiency.

We now have outlined the main ideas of the paper. The remainder of the discussion is devoted to filling them out with examples and explanations. The entire scheme has been developed as part of the operating system for the Berkeley Computer Corporation Model I. Since many details and specific mechanisms are dependent on the characteristics of the surrounding system and underlying hardware, we digress briefly at this point to describe them.

### *Environment*

The BCC Model I is an integrated hardware and software system designed to support a large number (up to 500) of time-sharing users. This system consists of two central processors, several small processors, a large central (core and integrated circuit) memory, and rotating magnetic memory. The latter contains more than $500 \times 10^6$ bytes, including approximately $12 \times 10^6$ bytes of drum having a transfer rate of more than $5 \times 10^6$ bytes per second.

The hardware allows each process more than 512k bytes of virtual memory. The central processors can accommodate operands of various sizes including 48- and 96-bit floating point numbers. The addressing structure allows characters, part-word fields and array elements to be referenced directly. The subroutine-calling instruction passes parameters and allocates stack space automatically. System calls are handled exactly like ordinary function calls; when arrays or labels are passed to the system they are checked automatically by the hardware so that they can be used by the system without further ado.

The memory management system organizes memory into *pages*. A page is identified by a 48-bit unique name which is guaranteed different for each page ever created in the system. Tables are maintained in the central memory which allow the page to be found in the various levels of the memory system. These tables are automatically accessed by the address mapping hardware the first time the page is referenced after the processor starts to run a new process. Thereafter its real core address is kept in fast registers. It is therefore unnecessary for any program other than a small part of the basic system to be concerned about the location of a page in the memory system; when it is referenced, it will be brought into the central memory if it is not already there. Extensive facilities are provided, however, to allow a process to control the level in the memory hierarchy of the pages it is interested in. The work of managing the memory is done by a processor with read-only program memory and data access to the central memory; this processor has a 100 ns cycle time, so that it can handle the large amount of computing required to keep up with demands placed on the memory system. Another small processor handles the remote terminals, which are multiplexed in groups of 20 to 100 at remote concentrators and brought into the system over high-speed lines.

Pages are grouped into *files*, which are treated as randomly addressable sequences of pages. The only mechanism provided to access the data in a file is to put a page of the file into the virtual memory of a process. Files and processes are named and have protection information associated with them.

*Domains in action*

Before plunging into a detailed analysis of capabilities and domains, we will look at some of the practical situations which these facilities are designed to serve. They all have the same general character: several programs with different privileges exist. Each program corresponds to one domain. Some of the domains control others, in the sense that the capabilities of a controlled domain are a subset of those of its controlling domain. As a first example, consider the command process CP of an operating system. This program accepts a command, perhaps from a remote terminal, and attempts to recognize it as a call on a program X which CP knows about. If it succeeds, CP calls on X for execution, passing it any parameters which were included in the command. To do this, CP must set up a suitable environment for X to function in. In particular, enough memory must be provided for X to run, X must be loaded properly, and suitable input/output must be available. When X is finished, it will return and CP can process a new command.

The key point is that we want CP to be protected from X, to ensure that the user's commands continue to be processed even if X has bugs. In particular, we want to be sure that

1. X does not destroy CP's memory or files, so that CP can continue to run when X returns.
2. CP can stop X if it goes wild. Usually we want the ability to set a time limit and also to intervene from the terminal.

In other words, we want CP and X to run in separate domains, as illustrated in Figure 1 (since this is an informal discussion, we do not trouble to distinguish carefully between the program X and the domain in which it runs). Here we have shown the call from CP

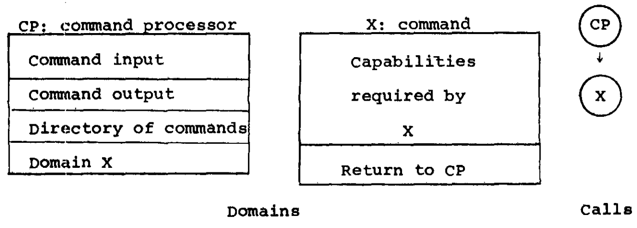


Figure 1—A command processor and its command

to X in two forms: in the picture on the right, and as a return capability in X. The reason for the capability is that X cannot return with a simple branch operation, since it would then be able to start CP running at any point, which would destroy the protection.

Suppose now that we want to allow X to get additional commands executed. X might, for example, be a Fortran compiler whose output must be passed through an assembler. A simple way to do this is to put the assembler input on a file called, say, FORTRANTEMP, and issue the command.

ASSEMBLE FORTRANTEMP, BINARY

This command is just a string, which can easily be constructed by the compiler X. To get it executed, however, X must be able to call CP. This situation is illustrated in Figure 2; note the call capability in X, which is quite different from the return capability. We are ignoring for the moment the question of how CP knows that X is authorized to call the assembler.

If the idea of the preceding paragraph is pursued, it suggests the value of being able to switch the source of command input and the destination of command output in a flexible way. By these terms we mean the

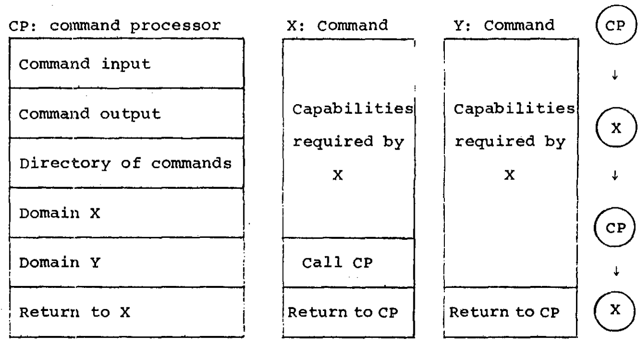


Figure 2—A recursive command processor

traffic between a program and the entity by which it is directed. In a time-sharing system this is normally a terminal at which the user is sitting; in a non-interactive system it will be a file of control cards. It is often desirable, however, to switch between the two, so that routine processing can be done automatically when the user's attention is elsewhere, yet he can regain control when things go awry. Again, it is not uncommon to wish to capture a complete record of a conversation between user and machine for later analysis and replay. More radical, it may be of interest to replace the user at his terminal with a program which can manipulate the strings of characters which constitute commands and responses. In this way major changes in the external appearance of a system can be obtained with little effort.

All of these things can be accomplished by giving interactions with the command I/O device the form of calls to a different domain which acts as a switch. A generalization to include the possibility of different command devices for different domains is easy. Thus, a user may initiate a program in a domain X which, while continuing to communicate with him, starts a

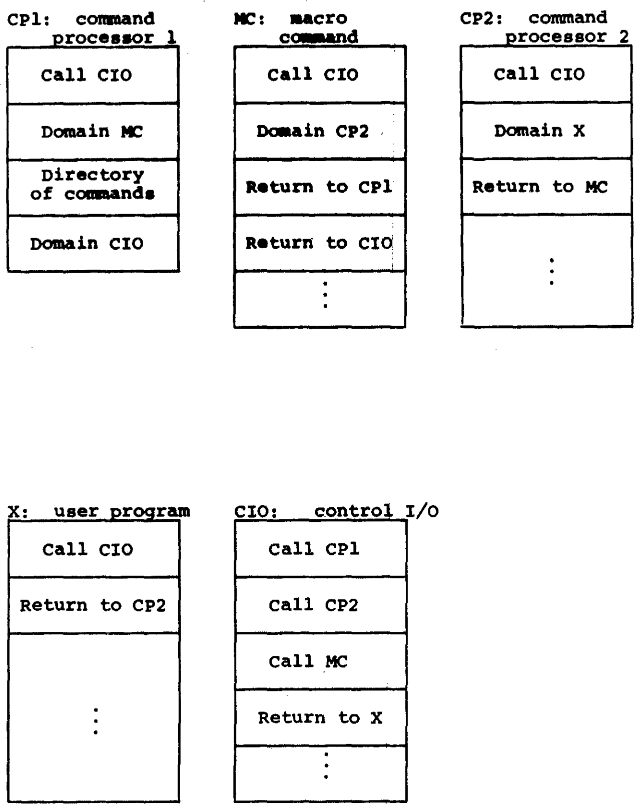
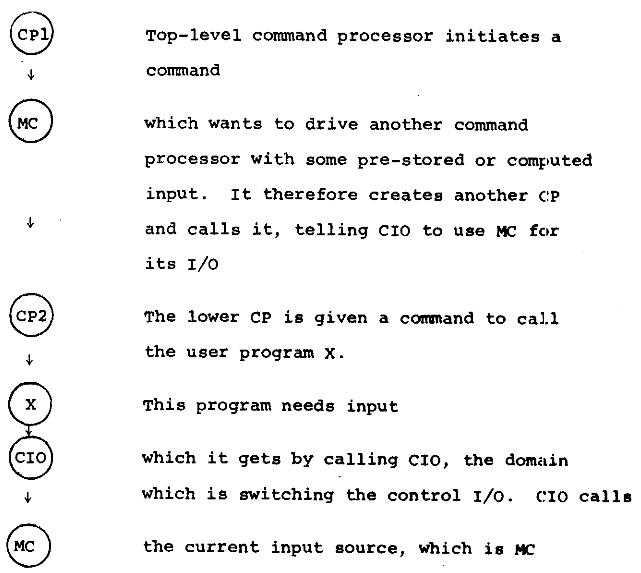


Figure 3a—Switchable control I/O—the domains



Top-level command processor initiates a command

which wants to drive another command processor with some pre-stored or computed input. It therefore creates another CP and calls it, telling CIO to use MC for its I/O

The lower CP is given a command to call the user program X.

This program needs input

which it gets by calling CIO, the domain which is switching the control I/O. CIO calls

the current input source, which is MC

Figure 3b—Switchable control I/O—the calls

subsidiary domain and feeds it commands. The subsidiary, unaware of the way in which it is being driven, may iterate the process by creating Z. The key fact which makes it all work is the isolation of one domain from others. Thus, Y may decide to close all its files without disturbing X, since Y has no way of even knowing about X's files, much less accessing them. Z, on the other hand, can be an open book to Y. Various aspects of the situation are illustrated in Figure 3.

This section concludes by analyzing a problem of great practical importance: how to construct a debugging system. This example is a good source of insights into the facilities required of a protection system because of the great variety of things which can be expected to go wrong during debugging. There are two domains, one for the debugger D and one for the program X being debugged. We of course want D to be protected from X. Equally important, we want X to be completely open to D, so that every object accessible to X is also accessible to D, and furthermore that D can find all the objects accessible to X as well as access them. Otherwise D will not be able to find out what X has done or to undo any damage. Furthermore, we want D to be able to imitate any actions which X can take, so that D can create suitable initial conditions for debugging parts of X. Thus, D needs operations which, given a capability for X, allow D to

find all the capabilities in X
copy capabilities between D and X
destroy capabilities in X
enter X at any point with any machine state

With these powers, D can also handle domains which X has created, since it can get hold of X's capabilities for them. Breakpoints can be inserted in X in the form of calls on D.

*Domains and capabilities*

### The nature of capabilities

As we have already said, a capability is a protected name of an object. When any object is created, a capability is created to name it; without the capability the object might as well not exist, since there is no way to talk about it. The capability may be thought of as an ordinary data item enclosed in a box which prevents tampering with the contents. Thus, for example, it may be convenient to make a capability for a file consist of simply the disc address of its index. This is entirely satisfactory, since programs which handle the capability cannot modify it. If they could, disaster would ensue, since any program could put any desired disc address into a file capability, and there would be no protection at all. If the machine hardware allows a word to be tagged so that it cannot be modified except by the supervisor, then we have precisely what we want for a capability. The situation is illustrated in Figure 4. It should be possible to load and store such a word (including the tag bits) in order to give programs the necessary freedom to manipulate the names of the objects they are working with.

If this kind of hardware is not available a different and potentially confusing implementation is required. The potential can be kept from realization by referring back to the "pure" implementation of the last paragraph. What is required is to hide the capabilities away in the supervisor and provide programs with unprotected names which can be used to refer to them. When a program running in domain D presents one of these names, it is necessary to check that it actually names a capability which belongs to D. This can easily

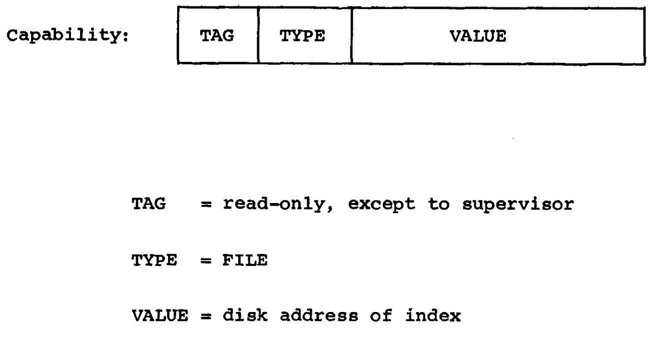
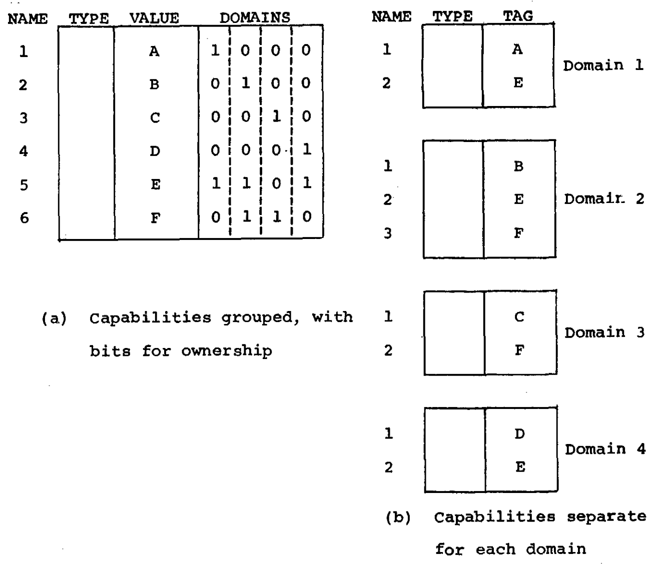

Capability:
TAG
TYPE
VALUE
TAG = read-only, except to supervisor
TYPE = FILE
VALUE = disk address of index

Figure 4—Structure of a capability


NAME
TYPE
VALUE
DOMAINS
(a) Capabilities grouped, with bits for ownership
NAME
TYPE
TAG
Domain 1
Domain 2
Domain 3
Domain 4
(b) Capabilities separate for each domain

| NAME | TYPE | VALUE | DOMAINS | | | |
|---|---|---|---|---|---|---|
| 1 | | A | 1 | 0 | 0 | 0 |
| 2 | | B | 0 | 1 | 0 | 0 |
| 3 | | C | 0 | 0 | 1 | 0 |
| 4 | | D | 0 | 0 | 0 | 1 |
| 5 | | E | 1 | 1 | 0 | 1 |
| 6 | | F | 0 | 1 | 1 | 0 |

| NAME | TYPE | TAG | |
|---|---|---|---|
| 1 | | A | Domain 1 |
| 2 | | E | |
| 1 | | B | Domain 2 |
| 2 | | E | |
| 3 | | F | |
| 1 | | C | Domain 3 |
| 2 | | F | |
| 1 | | D | Domain 4 |
| 2 | | E | |

Figure 5—Capabilities and unprotected names

be done, if there are $n$ such capabilities, by using numbers between 1 and $n$ for the names.[3] An attractive alternative, if domains can be grouped into larger units which share many capabilities, is to number the domains from 1 to $i$ and the entire collection of capabilities from 1 to $n$ and to attach a string of $i$ bits to each capability. Bit $d$ is on exactly when the capability belongs to domain $d$. Figure 5 illustrates.

A somewhat more expensive implementation is to search a table associated with the domain whenever an unprotected name is used. This scheme shares with the bit-string idea the advantage that it is easy for different domains to use the same names for the same object.

There are capabilities for all the different kinds of objects in the system. On the Model I these are

files
pages of memory
processes
domains
interrupt calls
terminals
access keys

### Domains and memory

The nature of a domain is considerably more dependent on the underlying system than is the case for capabilities, mainly because of the treatment of memory. From a purist's viewpoint, every access to a memory word is an exercise of a capability for that word. A more moderate position, and one which is quite feasible on suitable hardware, is to view each access as the exercise of a capability for a *segment* which contains the word.[2] The mapping hardware which implements segmentation is thus viewed as part of the capability system, and a satisfying unity of outlook is gained. Since a segment is identified by number, the preceding section applies. We shall not consider the formidable difficulties which arise if different domains use different names for the same segment.

If segments are accessed through capabilities like everything else, then a domain consists of nothing more than a collection of capabilities. On machines not equipped with the proper hardware a domain has an *address space* as well. In the Model I this is a list of the pages which occupy each of the 64 slots for pages in the 128k memory which is accessible to a user program.

It is also necessary to deal with the fact that the hardware does not allow one domain to access the address space of another one directly. This fact is of great importance when we consider how data is passed back and forth between domains, since it implies that arrays cannot be passed simply by specifying their addresses. It is therefore extremely convenient to include as part of a call the ability to pass scalar data items, and essential to include the ability to pass capabilities. From this foundation arbitrarily complex communication can be built, since capabilities for pages, files and domains can be passed. Thus, if an array needs to be passed as a parameter, it is sufficient to pass capabilities for the pages or file containing the array, together with its base address and length. The called domain can then put the pages into its address space and access the array. This is of course much less convenient than passing an entire segment as a parameter, but it is quite workable.

An alternative approach is to organize the hardware so that the address space of one domain is a subset to that of another. This eliminates all problems when the smaller one calls the larger, although it does not help at all when we want to share only part of the address space. A subset organization fits well with a linear or "ring"-like system[4] in which the domains are numbered, and the capabilities of domain i are a subset of those of domain i-1. As we shall see, there are good reasons for wanting a more flexible scheme, but for a great many applications a linear ordering is quite satisfactory. To allow these to be handled more efficiently, the Model I hardware breaks the address space of a process into three rings:

monitor
utility
user

in decreasing order of strength. The hardware enforces a restriction that addressing cannot go into a higher ring. It also provides protected entry points into the utility and monitor rings and automatically checks addresses passed into these rings as parameters to ensure that they are legal in the ring from which they came.

This simple hardware-implemented structure permits three domains to transfer control around among each other and to address each other's memory in a very convenient and efficient way. The price paid is a rigidity in structure, and a drastic incompatibility with the main, software-implemented domain mechanism. The incompatibility is resolved by requiring a change in ring to be reported to the software, except when the only processing to be performed before returning the original ring can be done with the capabilities of the original ring. Short calls thus remain cheap, while the overhead added to longer ones is not excessive.

## Domains and processes

The relationship between domains and processes is another area greatly influenced by the surrounding system. The logical nature of the two kinds of object allows a great deal of freedom: in fact, a domain has much the same appearance to a process that a segment of memory does. The storage for capabilities provided by a domain can accommodate many processes, and a single process can switch from one domain to another (subject to restrictions which are considered in the next section).

In the Model I, however, storage is allocated in 2k pages, and one of these, called the *context block*, is used to hold the system-maintained private data for each process. The cost of having a process is thus high, and there is considerable incentive to minimize the number of processes; usually one is enough per computation, if advantage is taken of the interrupt facilities described later. When the usage of space in the context block is analyzed, it turns out that there are only two items which would have to be duplicated to allow several processes to run with the same address space. These are a 14-word machine state and a stack used for local storage when the supervisor is executing in the process. This stack has a minimum of about 60 words and can grow to several hundred words at certain points during supervisor execution. It is therefore the main barrier to the existence of cheap processes. The problem can be greatly alleviated by allocating stack space dynamically at each function call and releasing it at each return, but this would require some major changes in system organization.

Although processes are expensive, domains are quite cheap, since the bit-string method is used to assign capabilities to domains. Each process in the Model I can have about a dozen domains associated with it. The process can run in any of its associated domains but in no others. This implies that two processes never run in the same domain.

In a system in which processes are cheap, it is possible to take an entirely different approach which encourages the creation of processes for every purpose. In such a system, parallel processing is of course greatly facilitated. In addition, free creation of processes can be used to give a somewhat different form to many of the facilities described in this paper.[3]

It is perhaps worthwhile to point out that a machine whose addressing is not organized around a stack or base registers cannot reasonably run several processes out of the same domain unless they are executing totally disjoint code, because of the problem of address conflicts.

*Transfers of control*

### Calls

The only reason for creating a domain is to establish an environment in which a process may execute with different protection than that provided by any existing domain. If this objective is to be fulfilled, transfers of control between domains must be handled with great care, since they generally imply the acquisition of new capabilities. If it is possible for a process running in domain X to suddenly jump into domain Y and continue execution at any arbitrary point, X can certainly induce Y to damage the objects accessible through Y's capabilities.

To provide an adequate mechanism for transfers between domains, we introduce the idea of a protected entry point or *gate*, and make the rule that transfer into a domain is normally allowed only at a gate. A gate is a new kind of capability which can be created by anyone with a capability for the domain. It specifies a location to which control is to go when the gate is used. Gates can be passed around freely like other capabilities, and each one may be viewed as conferring a certain amount of power, namely the power to accomplish whatever the routine entered by the gate is designed to do. With gates it is possible to selectively distribute the powers of a domain in a flexible way.

A transfer through a gate usually takes the form of a subroutine call; some provision must therefore be made for a return. It is not satisfactory to create another gate which the called process may return through, since he might save it away and use it to return at some later and unexpected time. Instead, the domain and location to return to are saved on a *call stack* in the supervisor, from which the return operation can retrieve them. It is possible to call a domain recursively with this mechanism, a feature which is generally desirable and also quite important for the trap and interrupt system about to be described.

In order to allow the stack to be reset in case of an error, or for any of the other reasons which prompt programmers to reset stacks, a jump-return ($n$) operation is provided which returns to the domain $n$ levels back. Protection is maintained by requiring the domain doing the jump-return to have capabilities for all the domains being jumped over.

### Traps

A *trap* is caused by the occurrence of some unusual event in the execution of the program which requires special handling, such as a floating point overflow, a memory protection violation or an end of file. When a trap occurs, it forces control to go to a specified place, where presumably a routine has been put to deal with the event. Whether any particular event causes a trap or simply sets a flag which can be tested by the program is a decision which should be under the programmer's control. Traps may be initiated by hardware (e.g., floating overflow) or may be artifacts of the software; as with most distinctions between hardware and software implementation, this one is of little importance, and we expect all traps to be transmitted to the program in the same form, regardless of their origin.

These are all obvious points which are generally accepted, and have even become embedded in the definition of PL/I. What concerns us here is the relationship between traps and domains, which is not quite so obvious. The basic problem is that the response to a trap must be made to depend on the environment in which is occurs. The occurrence of, say, a floating overflow is simply a fact, and has nothing to do with who is running. The action to be taken, on the other hand, is entirely a function of the situation. Consider the example in Figure 6. If a floating overflow occurs with the call stack in state (b), it is clear that

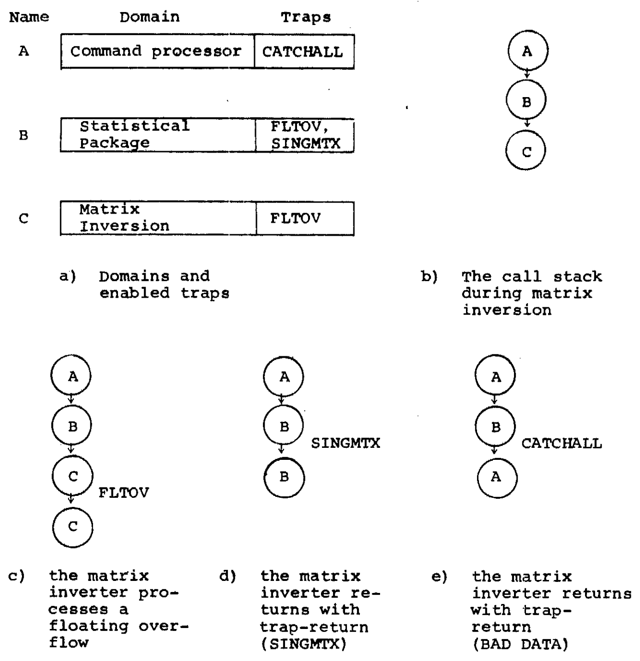

Name
Domain
Traps
A
Command processor
CATCHALL
B
Statistical Package
FLTOV, SINGMTX
C
Matrix Inversion
FLTOV
a) Domains and enabled traps
b) The call stack during matrix inversion
FLTOV
SINGMTX
CATCHALL
c) the matrix inverter processes a floating overflow
d) the matrix inverter returns with trap-return (SINGMTX)
e) the matrix inverter returns with trap-return (BAD DATA)

Figure 6—Traps and trapreturns

C should have the first chance to handle the trap. If it is not interested, the domain B which called it should have the second chance. In state (c), on the other hand, domain B should have the first chance, and then A. The reasons for this is that we do not wish to give up control to a weaker domain when a trap occurs.

The idea is then the following: Each domain is considered to have a *father*. When a trap occurs, it is first directed to the domain S which is running. If S does not have the trap enabled, the father of S is tried in the same way. If no one can be found to handle the trap, there are two possibilities:

ignore it;
generate a catchall trap which any domain that lacks a father is forced to handle.

If a domain T is found with the trap enabled, it is called with the name of the trap as argument. It can then return and allow execution to proceed if it is able to clear things up. Alternatively, it can do a jump-return to someone farther back on the call stack if it finds the situation to be hopeless. An important property of this scheme is that the trap routine can do arbitrarily complex processing without disturbing the situation at the time of the trap.

Conceptually, we wish to think of traps as identified by symbolic names. Each domain must then include a list of names of the traps it has enabled. Corresponding to each hardware-generated trap is a standard name. Software-generated traps can use any names, including the ones for hardware traps. This makes it easy for a subroutine to simulate the occurrence of a hardware condition which it may not be convenient to produce.

A simple extension of the return operation to a *trap-return* allows a routine to signal an error without leaving any traces of itself; the trap-return does a return and immediately causes the specified trap, without allowing any execution beyond the return point. The domain which handles the trap then sees it as having occurred in the calling routine, which is exactly what is wanted. Thus in Figure 6 we have a matrix inversion routine which processes its own floating overflows, but reflects two other conditions to its caller with trap-return. Another useful convention is to disable the trap when it occurs. This makes it much less likely that the program will get into a loop, especially for such traps as illegal instruction and memory protection violation.

## Interrupts

There remains one more way to cause a transfer between domains: the occurrence of an *interrupt*. This is not intended to be the normal mechanism for communication between cooperating processes; the basic block and wake-up mechanisms[5] are expected to perform that function. There are times, however, when it is desirable to force a process to do something, even if it is not paying attention. Two obvious reasons for this are:

a *quit* signal from the terminal, which indicates that the user wants to regain control over a process which has gone into a loop, or perhaps simply become unnecessarily wordy;

the elapse of a certain amount of time, which has much the same meaning.

The action required in these two cases is different. When a timer interrupt is requested (and there may be two kinds, for real time and CPU time) the desired action is usually to call a specific domain, often the one which is setting the timer. If another domain wants a timer, it will use one which is logically different. The user's quit signal, on the other hand, is context dependent like a trap; the desired action is a function of the routine which is running when the signal arrives. Thus an iterative root-finder may interpret a quit as an indication that the solution is accurate enough, but the debugging system under which it may be running will curtail its printing when it sees a quit and await a new command. This analysis suggests a simple implementation: convert the quit into a trap from the currently executing domain. Each interrupt, then, will give rise to a call or a trap, depending on its type as declared by the programmer.

Even when we see how to convert them into operations within the process, interrupts still present one serious problem which does not arise in the handling of traps. This is the fact that a program occasionally needs to be allowed to compute for a while without losing control. Usually this happens when modifications are being made to a data base; if a quit signal should appear or a timer run out halfway through this operation, the data is left in a peculiar state. The obvious solution is to allow a process to become non-interruptible for a limited period of time. The function of the limit is to prevent the process from getting into a state from which it cannot be retrieved; exceeding it is a programming error and always causes the process to become interruptible again and an error trap to occur, regardless of whether an interrupt is actually pending. The limit is properly measured in real time, since its primary purpose is to put a bound on the frustration of the user at his console.

Non-interruptibility is a process-wide condition. It must be possible, however, for a newly-called domain to extend the limit exactly once, so that it can function properly even though its caller is about to exceed his limit. The limit is thus part of a call stack entry. When a return occurs, the old limit comes back into force, and an immediate trap may occur if it has been exceeded.

Table I summarizes the operations connected with transfers of control between domains.

TABLE I—Operations for transfers

| *Operation* | *Arguments* |
|---|---|
| Call | Gate, Parameters |
| Return | Parameters |
| Jump | Gate, Parameters |
| Jump-return | Depth, Parameters |
| Trap | Trap number |
| Trap-return | Trap number |

## *Proprietary programs*

The remainder of this paper deals with the protection problems introduced when objects are allowed to have external, mnemonic names. The examples in this section are intended to introduce this subject, and are also of interest in their own right. Suppose then that a user U has a program executing in domain P and wishes to perform a circuit analysis. P has generated the input data for the analysis, and intends to use the results for further calculation. Within the system M on which P is running, some user V has written a suitable analysis program A which he has offered for sale, and U has decided to use V's program. It happens that U and V are competitors.

Both users in this situation have selfish interests to protect. First, and most obvious, V does not want his program stolen. He therefore insists that while it is executing U must not be allowed to read it. Equally important, however, is the fact that U does not want V's program to be able to read the calling program P and its data; although U may not be trying to market P, it, and especially its data, contain valuable information about U's current development work which must be kept from competitors. The relationship between U and V, and between their programs P and A, is therefore one of mutual suspicion. Each is willing to entrust the other with just enough information to allow the circuit analysis to be completed, and no more. The system must support this requirement if it is to be a suitable vehicle for selling programs.

Furthermore, care must be taken beyond the programs. While P is running it needs the ability to access U's files by name, to read input data and record results. This privilege must certainly not be extended to A, since it can learn even more about U's secrets by examining his files than by looking at his program, not to mention the possibility of modifying them. On the other hand, A may need access to V's files to obtain data for the analysis and to collect statistics and accounting information; this access must not be available to P. The protection mechanisms must therefore provide for isolating P and A at the level of file naming as well as on the lower levels which have been the subject of this paper so far.

What is required then is a system facility something like this. V establishes A as a *proprietary program*, specifying the file on which it resides. Another user's program P may then ask the system to *attach* this file. To do this, the system creates a new domain A, installs the program in it, provides it with some storage, and returns to P a gate into A. When P wants to call A, he uses the gate and passes whatever parameters he thinks are needed for A to function. When A is finished, he returns. The protection mechanisms we have been discussing prevent undesired interference between P and A. Safeguards for the files are discussed below.

The example above is one of a great variety of similar situations. The system itself creates many of them. A LOGOUT command, for example, requires special access to accounting files and to capabilities for destroying a process, but it would be nice to call it with the standard command processor. Similarly, driving a special peripheral like a printer requires special capabilities. If a company maintains a large data base, it may wish to give different classes of users access to different parts of it by allowing them to call different accessing programs. These and many other applications fall within the general outline established by our proprietary program example. We now proceed to consider how to handle the file naming problems it presents.

*External names*

Table II lists the goals of a naming system for objects, and indicates some of the distinctions between the use of capabilities in names which have been discussed in previous sections, and the use of *external names,* which are strings of characters such as 'FILE1' or 'CIRCUIT'. In summary, it says that capabilities are very convenient for use by a program, since they are cheap and self-validating. On the other hand, they are very bad for people, since they cannot be typed in or remembered. Names for people should also have the property that the same name can refer to many different objects, the distinctions to be made by context. Thus, Smith's file 'ALPHA' is not the same as Jones' 'ALPHA'.

TABLE II— Goals of a naming system for objects

| Goal | Achieved by Capabilities | Achieved by external names |
|---|---|---|
| Names are mnemonic | | X |
| Names can be relative to other names | | X |
| Names can be used externally | | X |
| Possession of name authorizes access | X | |
| Names are cheap to use | X | |
| Names can be manipulated by programs | X | X |

Techniques for achieving all these goals are well known. They depend on the introduction of a new kind of object called a *directory,* which consists of pairs: <external name, capability>, and an operation of *opening* an object by supplying the name to obtain the capability. Since the external name is interpreted relative to a directory, there is a suitable basis for establishing the context of a name. A tree-structured naming system is implicit in the scheme, because directories are themselves objects accessed by capabilities. It is now easy to see how a program in a domain D accesses the objects belonging to owner U. When D is created, it is supplied with a capability for U's directory, which it simply exercises.

There is more controversy over the proper methods of accessing objects belonging to other users. A popular approach is to use passwords: a public read-only directory is filled with capabilities for all other directories which allow the objects in them to be accessed provided a correct password (usually different for each object) is supplied as part of the opening operation. This method is not satisfactory. First, it is inconvenient, since it requires the person accessing the file to remember the password. Second, it is insecure. If he writes the password down, or includes it in a program, the possibility increases that it will become known. It is bad enough to have to use a password to obtain entry to the system, but at least only one password is involved, it is used only once per session, and it can be changed, if need be after each session, without too much fuss. None of these things is true of passwords attached to files: there are many of them, many people need to know them, and one must be used each time a file is opened. This scheme has no advantage except economy of implementation.

A method based entirely on capabilities suffers only one of these drawbacks: it is inconvenient, but secure. It is also, however, quite complex. The idea is that if a file (or anything else) is to be shared, a capability for it should be passed from its owner to those who wish to share it. The problem is that a capability, being a protected object, must be passed through protected channels; it cannot be sent in a letter, even a registered letter. The solution is illustrated in Figure 7. Every user has (at least) two directories, a private one which he works with, and a *transfer directory.* The public directory PUB, for which every user has a read capability, contains write capabilities for all the transfer directories. The object is to move the capability for X from PDA to PDB. Proceed as follows:

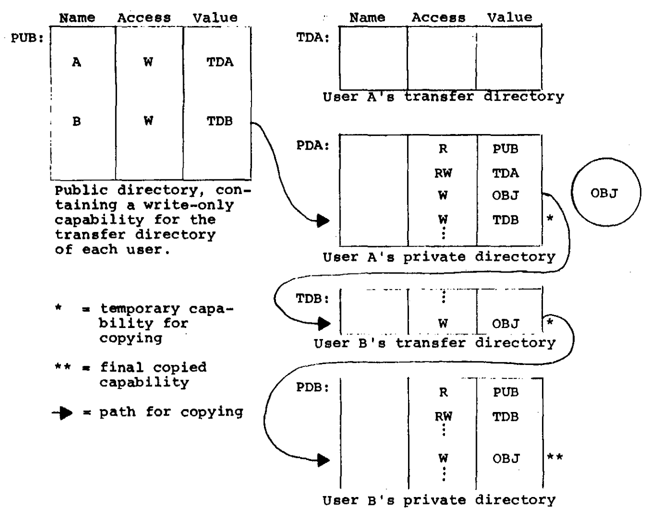

Name Access Value
PUB:
A W TDA
B W TDB
Public directory, containing a write-only capability for the transfer directory of each user.
TDA:
User A's transfer directory
PDA:
R PUB
RW TDA
W OBJ
W TDB *
User A's private directory
OBJ
TDB:
W OBJ *
User B's transfer directory
PDB:
R PUB
RW TDB
W OBJ **
User B's private directory
* = temporary capability for copying
** = final copied capability
= path for copying

Figure 7—Sharing capabilities without access keys

A moves a capability for TDB into PDA
Using it, A moves his capability for X to TDB
B moves the capability for X from TDB to PDB

Since only B can access TDB, security is preserved. A malicious user can confuse things by writing random capabilities into the TDs, but it is easy for B to check that he has gotten the right thing. Furthermore, if X is a directory, future communication can be carried out quite conveniently, since A and B can then communicate through X without any worries about outside interference.

A much better method is based on the simple idea of attaching to a directory entry a list of the users who are allowed to access it; with each user we can also specify options, so that Rosenkrantz may be granted write access to the file while Guildenstern can only read it. This scheme, which was first used in CTSS,[1] has two drawbacks. The first is that if the list of users who are authorized to access a file is long, it takes a lot of space to store it; this problem is especially annoying if there are several files to be accessed by the same group of users. The second drawback is that there is no provision for giving different kinds of access to different domains of a computation. Both difficulties can be overcome in a rather straightforward manner.

Before we pursue this point, it is important to notice why the difficulty encountered above in the capability-passing scheme does not arise here. We can think of the computation of a logged-in user as possessing a special kind of capability which identifies it as belonging to him. If SMITH is the user, we will refer to this capability as SMITH*, meaning that the string

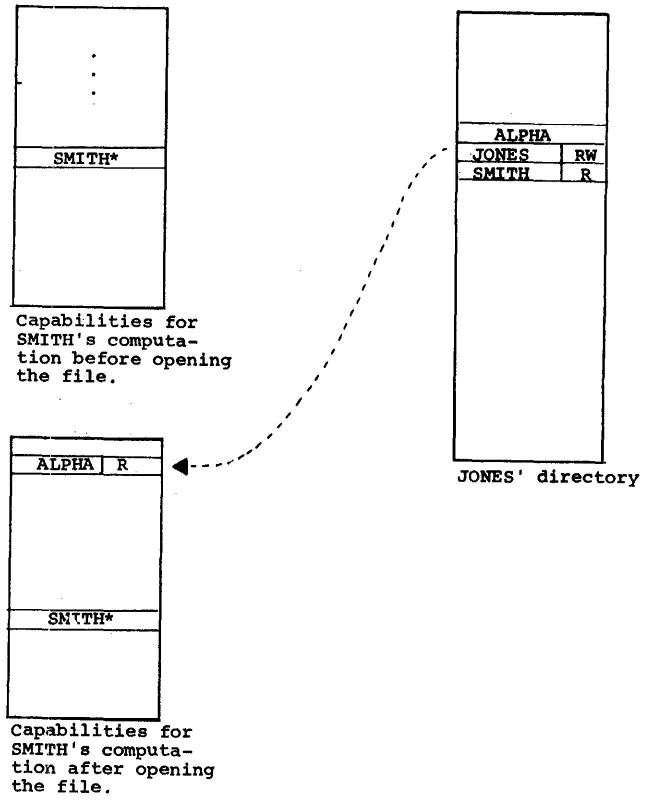

SMITH*
Capabilities for SMITH's computation before opening the file.
ALPHA R
SMITH*
Capabilities for SMITH's computation after opening the file.
ALPHA
JONES RW
SMITH R
JONES' directory

Figure 8—Use of access keys

'SMITH' has been enclosed in a tamper-proof box. When JONES wishes to give SMITH access to his file ALPHA, he puts the name SMITH on the access list; JONES can do this since he has a capability for ALPHA. When a computation presents the capability SMITH*, the system observes that the string (or user number) which is the contents of the capability matches the string on the access list and grants the access. At no time is it necessary for JONES to have SMITH* in his possession. He needs only the name SMITH which, since it is not a protected object, can be communicated to him by shouting across the room. Figure 8 illustrates.

To generalize the method we need two ideas. One is that of an *access key*. This is an object (i.e., it can be referenced only by using a capability) which consists simply of a bit string of modest length, long enough that the number of different access keys is larger than the number of microseconds the system will be in existence. Any user may ask the system for a new access key; the system will create one never seen before and return a capability for it. The object SMITH* mentioned in the last paragraph is an example of an access key; one is kept for each user in the system. Since an access key is an object, capabilities for it appear in the directories and are protected exactly as is done for any other object (since the access key is a small object, it may be convenient for the implementation not to give it any existence independently of the capabilities for it, i.e., to make the value of the capability the object itself, rather than a pointer to it as in the case of files). To give a group of users access to some files, all we have to do is distribute a new access key GROUP* to the users and put GROUP on the access list for each file. The distribution is accomplished by creating GROUP* and putting all the users on its access list; once they have copied it into their directories they can be removed from the access list, so that no space need be wasted. In practice, as we have pointed out, numbers of perhaps 64 bits would be used instead of strings like 'GROUP'.

The second idea is not new at all. It consists of the observation that since an access key is just an object, different domains can have different access keys and hence different kinds of access to the file system. Thus, for example, a user's computation may be started with two domains, one for his program with his name as access key, and the other for system accounting with an access key which allows it to write into the billing files. With a single suitable access key, a domain can easily get hold of an arbitrarily large collection of other objects which are protected by other keys, since the first key can be used to obtain other keys from the directory system.

## SUMMARY

We have described a very general scheme for distributing access to objects among the various parts of a computation in an extremely specific and flexible way. The scheme allows two domains to work together with any degree of intimacy, from complete trust to bitter mutual suspicion. It also allows a domain to exercise firm control over everything created by it or its subsidiaries.

## REFERENCES

1 P A CRISMAN editor
*The compatible time-sharing system: A programmer's guide*
MIT Press 2nd ed Cambridge Mass 1965

2 J P DENNIS
*Segmentation and the design of multi-programmed computer systems*
J ACM Vol 12 Oct 1965 589

3 J B DENNIS E C VAN HORN
*Programming semantics for multiprogrammed computation*
CACM Vol 8 No 3 March 1966 143

4 R M GRAHAM
*Protection in an information processing utility*
CACM Vol 11 No 5 May 1968 368

5 B W LAMPSON
*A scheduling philosophy for multi-processing systems*
CACM Vol 11 No 5 May 1968 347

6 B W LAMPSON et al
*A user machine in a time-sharing system*
Proc IEEE Vol 54 No 12 Dec 1966

# The ADEPT-50 time-sharing system

*by* R. R. LINDE and C. WEISSMAN

*System Development Corporation*
Santa Monica, California

and

C. E. FOX

*King Resources Company*
Los Angeles, California

## INTRODUCTION

In the past decade, many computer systems intended for operational use by large military and governmental organizations have been "custom made" to meet the needs of the particular operational situation for which they were intended. In recent years, however, there has been a growing realization that this design approach is not the best method for long term system development. Rather, the development of general purpose systems has been promoted that provide a broad, general base on which to configure new systems. The concepts of time-sharing and general-purpose data management have been under development for several years, particularly in university or research settings.[1,2,3] These methods of computer usage have been tested, evaluated, and refined to the point where today they are ready to be exploited by a broad user community.

Work on the Advanced Development Prototype (ADP) contract was begun in January 1967 for the purpose of demonstrating—in an operational environment—the potential of automatic information-handling made possible by recent advances in computer technology, particularly advances in time-sharing executives and general-purpose data management techniques. The result of this work is a large-scale, multi-purpose system known as ADEPT, which operates on IBM system 360 computers.*

The entire ADEPT system is now being used at four field installations in the Washington, D. C. area, as well as at SDC in Santa Monica. The system was installed at the National Military Command System Support Center in May 1968, at the Air Force Command Post in August 1968, and at two other government agencies in January 1969. These four field sites collectively run ADEPT from 80 to 100 hours per week, providing a total of some 2000 terminal hours of time-sharing service monthly to their users.

The ADEPT system consists of three major components: a time-sharing executive; a data management system adapted from SDC's Time-Shared Data Management System (TDMS) described by Bleier,[4] and a programmer's package. This paper deals exclusively with the ADEPT Time-Sharing Executive, and particularly with the more novel aspects of its architecture and construction. Before examining these aspects it will be instructive if we review the basic design and hardware configuration of the system.

### *A general purpose operating system*

The ADEPT executive is a general-purpose time-

* Development of ADEPT was supported in part by the Advanced Research Projects Agency of the Department of Defense.

sharing system. The system operates on a 360 Model 50 with approximately 260,000 bytes of core memory, 4 million bytes of drum memory, and over 250 million bytes of disc memory, shown graphically in Figure 1 and schematically in the appendix. With this machine configuration, ADEPT is designed to provide responsive on-line interactive service, as well as background service to approximately 10 concurrent user jobs. It handles a wide variety of different, independent application programs, and supports the use of large random-access data files. The design—basically a swapping system—provides for flexibility and expansion of system functions, and growth to more powerful models in the 360 family.

ADEPT functions both as a batch processor (whereby jobs are accumulated and fed to the CPU for operation one by one) and as an interactive, on-line system (in which the user controls his job directly in real time simply by typing console requests).

Viewed as a batch system, ADEPT allows jobs to be submitted to console operators or submitted from consoles via remote batch commands (remote job entry). In either case, jobs are "stacked" for execution by ADEPT in a first-in/first-out order. The stack is serviced by ADEPT as a background task, subject to the priorities of the installation and the demands of "foreground" interactive users. Viewed as an interactive system, ADEPT allows the user to work with a typewriter, allowing computer-user dialog in real time. Via ADEPT console commands, the user identifies himself, his programs, and his data files, and selectively controls the sequence and extent of operation of his job in an ad lib manner. A prime advantage of the interactive use of ADEPT is that the system provides an extendable library of service programs that permit the user to edit data files, compile or assemble programs, debug and eliminate program errors, and generally manage large data bases in a responsive on-line manner.

## *System architecture*

The architecture of the ADEPT executive is that of the "kernel and the shell". The "kernel," referred to as the Basic Executive (BASEX), handles the major problems of allocating and scheduling hardware resources. It is small enough to be permanently resident in low core memory, permitting rapid response to urgent tasks, e.g., interrupt control, memory allocation, and input/output traffic. The "shell," referred to as the Extended Executive (EXEX), provides the interface between the user's application program and the "kernel". It contains those non-urgent, large-

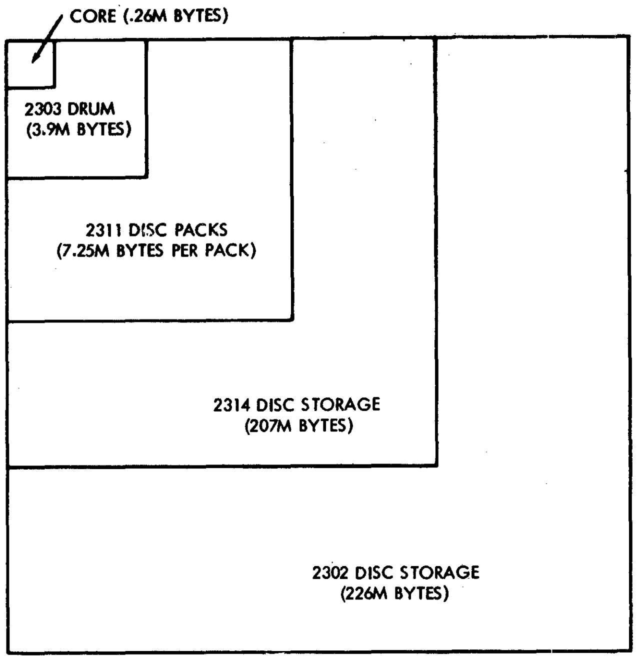

CORE (.26M BYTES)
2303 DRUM (3.9M BYTES)
2311 DISC PACKS (7.25M BYTES PER PACK)
2314 DISC STORAGE (207M BYTES)
2302 DISC STORAGE (226M BYTES)

Figure 1—Relative capacity of various ADEPT direct-access storage media available in less than 0.2 seconds. The initial system that operates at SDC utilizes core, 2303 drum, 2311 and 2314 disc packs, and 2302 disc storage. The NMCSSC system utilizes 2314 disc storage in lieu of 2311 or 2302 discs. The architecture of the ADEPT executive is such that it permits any combination of the above types of disc storage in varying amounts

task extensions of the basic "kernel" processes that are user-oriented rather than hardware-oriented; they may, therefore, be scheduled and swapped.

The version of the ADEPT time-sharing system, thus far developed has multiple levels of control beyond the two-level "kernel-shell" structure—i.e., it can be thought of figuratively as an "onion skin". Figure 2 shows these relationships graphically.

Beyond EXEX, "object systems" may exist as subsystems of ADEPT (developed by the user community without modification to EXEX or BASEX), thus further distributing and controlling the system resources for the object programs that form still another level of the system. The design ideas embodied in ADEPT parallel those of Dijkstra,[5] Corbato,[6] and Lampson,[7] but differ in techniques of implementation.

The ADEPT Basic Executive operates in the lower quarter of memory, thereby providing three quarters of memory for user programs. With the current H core configuration, ADEPT preempts the first 65,000 bytes of core memory, the bulk of which is dedicated to BASEX; EXEX must then operate in user memory

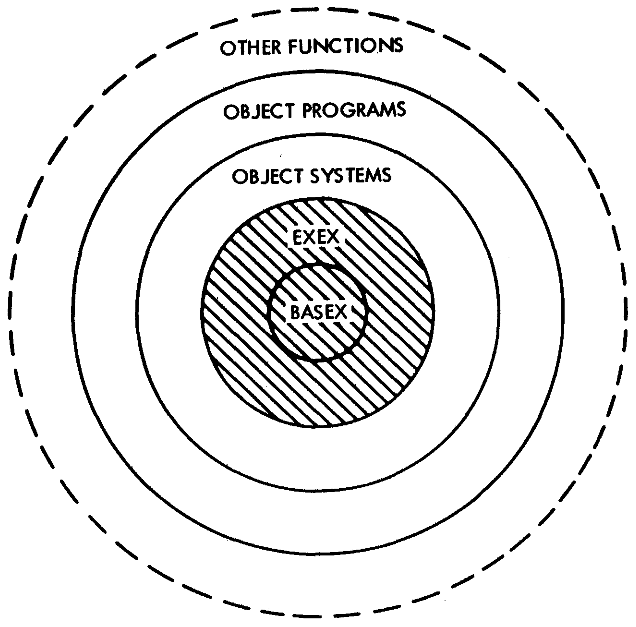


Figure 2—Multiple levels of control in ADEPT

in a fashion similar to user programs. ADEPT is designed to operate itself and user programs as a collection of 4096-byte pages. BASEX is identified as certain pages that are fixed in main storage and that cannot be overlayed or swapped. EXEX and other programs are identified as sets of pages that move dynamically between main storage and swap storage (i.e., drum). It is necessary to maintain considerably more descriptive information about these swappable programs than about BASEX. This descriptive information is carried in a set of system tables that, at any point in time, describe the current state of the system and each program.

ADEPT views the user as a job consisting of some number of programs (up to four for the 360/50H configuration) that were loaded at the user's reouest. These programs may be independent of one another or, with proper design, different segments of a larger task. Implicitly, EXEX is considered to be one of these programs. To simplify system scheduling, communication, and control, only one program in the user's set may be active (eligible to run) at a time. When ADEPT scheduling determines that a job may be serviced, the current job in core is saved on swap storage, and the active program of the next job is brought into core from swap storage and executed for a maximum period of time, called a quantum. The process then repeats for other jobs. Figures 3 and 4 schematically depict these relationships.

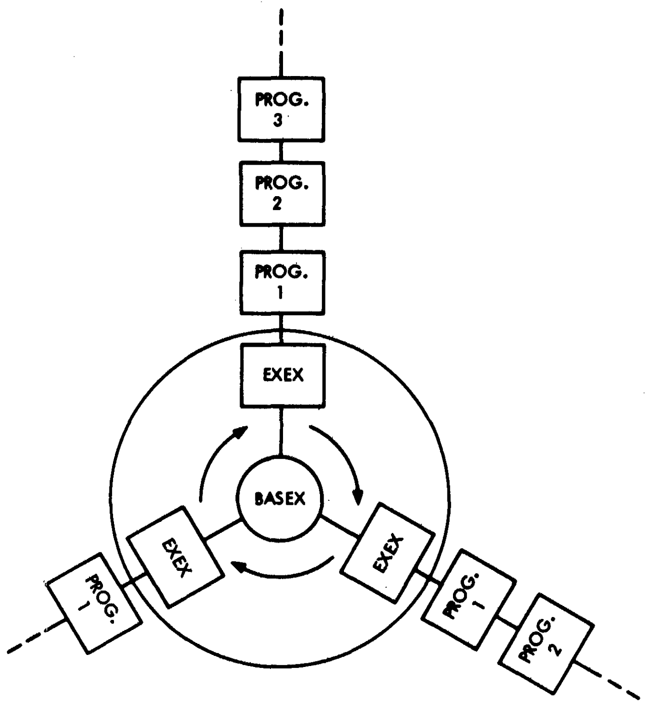


Figure 3—Simple commutation of users programs. This figure illustrates the relationship between user's programs' EXEX and BASEX. Each spoke represents a user's job, with his EXEX providing the interface between BASEX and the hardware resources. The maximum number of interactive job the IBM 360/50H configuration is ten.

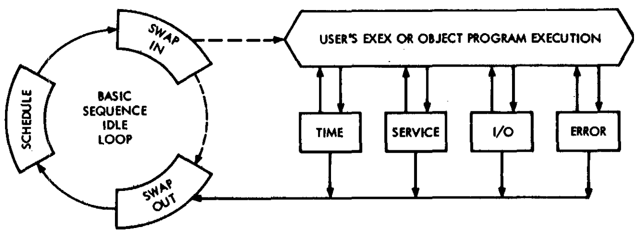


Figure 4—ADEPT's basic sequence of operation. This figure shows the basic operating system cycle: idle loop is interrupted by an external interrupt (an activity request); a program is scheduled, swapped into core from the drum, and executed escape from the execution phase occurs when quantum termination condition (e.g., time expiration, service or I/O call, error condition) is met; the program is then swapped out and control is returned to the idle loop (if no other programs are eligible to be scheduled).

## Basic executive (BASEX)

Table I lists the BASEX components and their general functions as of the eighth and latest executive release. These basic system components form an integrated, non-reentrant, non-relocatable, permanently-resident, core memory package 16 pages long (each page is 4096 bytes). They are invoked by hardware interrupts in response to service requests by users of terminals and their programs. Note the division of input/output control into cataloged (SPAM and IOS), terminal (TWRI), and drum (BXEC) activities to permit local optimization for improved system performance.

TABLE I—Basic executive components

| *Component* | *Function* |
|---|---|
| ALLOC | Drum and core memory allocation. |
| BXBUG | Debugger for executive programs. |
| BXEC | Basic sequence and swap control. |
| BXECSVC | SVC handlers for WAIT, TIME, DEVICE, STOP AND DISMISS calls. |
| EXEX | Linkage routines for EXEX (BASEX/EXEX interfaces); also services commands DIALOFF, DIALON. |
| INTRUP | First-level interrupt control. |
| IOS | Channel-program level input/output supervisory control. |
| RECORD | Records SVC, interrupt activity in BASEX. |
| SKED | Scheduler. |
| SPAM | Input/output access methods to cataloged storage. |
| TWRI | Terminal input/output control. |
| System Tables | Resident system data areas for communication table (COMTAB), logged-in user's table (JOB), loaded programs table (PQU), drum and core status tables (DSTAT, CSTAT), and a variety of other tables. |

## Extended executive (EXEX)

Unlike the tight, closed package of integrated BASEX components, EXEX is a loose, open-ended collection of semiautonomous programs. Table II lists this collection of programs. EXEX is treated by BASEX as a user program, with certain privileges, and each user is given his own "copy" of the EXEX. It is transparent to the user that EXEX is reentrant

TABLE II—Extended executive components

| *Component* | *Function* |
|---|---|
| AUDIT | Maintains a real-time recording of all security transactions as an accountability log. |
| BMON | Batch monitor for control of background job execution. |
| CAT | Cataloger for file storage access control; also services FORGET command. |
| DTD | Transfers recording information from drum to disc. |
| DBUG | Debugger for non-executive (user) programs. |
| LOGIN | User authentication and job creation. |
| SERVIS | Library of service commands that are reentrant, interruptible and scheduled: APPEND, CHANGE, CREATE, CYLS, DELETE, DRIVES, INIT, LISTF, LISTU, LOAD, LOADD, LOAD and GO, OVERLAY, REPLACE, RESTORE, RESTORED, SAVE, SEARCH, VARYOFF, VARYON. |
| RUN | Remote batch job submission control servicing commands RUN and CANCEL. |
| XXTOO | Library of small, fast, executive service commands: CPU, BGO, BQUIT, BSTOP, DIAL, DRUMS, GO, LOGOUT, QUIT, RESTART, SKED, SKEDOFF, STATUS, STOP, TIME, USERS. |
| SYSDEF | Defines input/output hardware configuration at time of system start up. |
| SYSLOG | Defines authorized user/terminal security profiles at time of system start up. |
| TEST | Initializes system tables at time of system start up. |
| SYSDATA | Non-resident, shared, system data table for dial messages and other common data, e.g., lists of all logged-in users; other non-resident, job-specific tables also exist, e.g., job environment page, push-down list data page. |

and is being shared with other users, except for its data space. Each job has its own "machine state" tables saved in its unique set of environment pages. This structure permits flexible modification and orderly system expansion in a modular fashion. EXEX is always scheduled in the same way as other user programs.

Though EXEX components are, in large part, non-self-modifying reentrant routines and thus, could at small cost, be relocatable; neither user programs nor EXEX components are relocated between swaps. The lack of any mapping hardware on the IBM 360/50 and the design goal and knowledge that most user programs would be of maximum size made unnecessary a software provision to relocate programs dynamically. User programs may be relocated once at load time, however.

*Communication and control techniques used in ADEPT*

Communication is the generic term used to cover those services that permit two (or more) programs to intercommunicate, be they system program, user program, or both. From this communication vantage point we shall examine the connective mechanism used between the Basic and Extended Executives; the techniques that allow components within the EXEX to make use of one another; and the system design that permits an object program to control its own behavior as well as to communicate with the system and with other object programs.

### The ADEPT job or process

Before we discuss the system mechanics, let us examine how the system treats each user logically. A user in the system is assigned a job number. Each job in the system may be viewed as a separate *process*, and each process is, by definition, independent of all other processes running on the machine. A process— or job— is not a program. It is the logical entity for the execution of a program on the physical processor, and it may contain as many as four separate programs. A program consists of the set of machine instructions swapped into the processor for execution, and the Extended Executive is one of these programs.

The ADEPT executive requires a large number of system tables to permit Basic and Extended Executive communication. Conceptually, the use of descriptive tables defining the condition of a user's process is analogous to the state vector (or state word) discussed by Lampson and Saltzer.[8,9] That is, the collection of information contained by these tables is sufficient to define an inactive user's process state at any given moment. By resetting the central processor from the state vector, a user's job proceeds from an inactive to an active state as if no interruption had occurred. The state vector contains such items as the program counter, the processor's general registers, the core and drum map of all the programs in the job, and the peripheral storage file data. All of the collective data for each program or task in the process are contained in the state vector.

### Basic and extended executive communication

Each ADEPT user (i.e., any person who initiates some activity within the system by typing in commands) is given a job number and assigned an entry in the JOB table. The JOB table contains the system's top-level bookkeeping on user activity. It contains the user's identification, his location, his security clearance, and a pointer to his program queue. Each user is assigned one entry, or JOB, in the table. Associated with each JOB are the one or more programs that the user is running.

Top-level bookkeeping on programs is contained in the Program Queue (PQU) table. Each PQU entry contains a program identification and some (but not all) information that describes that program in terms of its space requirements, its current activity, its scheduling conditions, and its relationship to other programs in the PQU that belong to the same JOB. The detailed descriptive information and the status of each JOB and its programs are carried in the swappable environment space.

The environment pages (there can be as many as four) comprise a number of separate tables that contain such information as the contents of the general registers, the swap storage page numbers where the balance of the program resides, the program map, and lists of all active data files. A single environment page (or pages) is shared by all programs that belong to the same JOB (user). The system design allows for environment page overflow at which time additional pages are assigned dynamically. The environment pages, PQU table, JOB table, and data pages comprise the state vector of the user's job.

To permit storage of "global" system variables, and to allow system components to reference system data that may be periodically relocated, there exists a system communication table, which resides in low core so that it can be referenced without loading a base register.

The IBM 360 supervisor call (SVC) is used exclusively by EXEX components and object programs to request BASEX services. Though additional overhead is incurred in the handling of the attendant interrupt, the centralization of context switching provided is of considerable value in system design, fabrication, and checkout.

### Extended executive communication

An EXEX may make use of another EXEX function by use of the SVC call mechanism. To support the recursive EXEX, an additional SVC processing routine is required to manage the different recursive contexts. This routine, called the SVC Dispatcher, processes calls from user and EXEX functions alike, manages a swappable data page, and switches to an interface linkage routine. The data page contains a system communication stack that consists of a program's general registers and the Program Status Word at the time of the SVC. This technique is analogous to the push-down logic of recursive procedure calls found in ALGOL or LISP language systems. The stack provides a convenient means of passing parameters between routines in the EXEX. Since each job has its own unique data page and environment page, EXEX is both recursive and reentrant.

The environment status table (ESTAT) contains the swap and core location for each component in the EXEX and for each program in the job. It resides in the job environment page. When an EXEX service is requested, only that particular EXEX program is brought in from swap storage, rather than the full service library. The interface linkage routine provides this management function; it lies as a link between the SVC Dispatcher and the particular EXEX function. The interface routine picks up necessary work pages for the EXEX component involved and branches to that component after it is brought into core. The interface routine maintains a separate push-down stack of return addresses providing the means for the EXEX component to properly exit and return control to its interface routine and then to the system.

The EXEX component called may make additional EXEX SVC calls before exiting. To provide correct work page allocation during recursive calls, the interface routine also saves the work page core and drum page addresses in the push-down stack. Upon completion of a call, the EXEX component returns to its interface routine; the interface routine releases all allocated work pages to the system and branches to a common unwind procedure.

The unwind procedure, like the SVC Dispatcher, is simply a switching mechanism. It determines, via the stack, whether to return to a still higher level EXEX function, or to turn the EXEX off and exit to the Basic Sequence. This recursive/reentrant control is the most complex portion of ADEPT and is the "glue" that binds BASEX and EXEX together. Figure 5 illustrates the recursive process.

### Object program communication

One of the more stringent services required of an operating system is the rapid interchange of large quantities of data between object programs. The interchange of even simple arrays, matrices, and tables via stack parameters or a common file suffers from the inadequacy of limited capacity or extensive I/O time. Many operating systems ignore this requirement, thereby restricting the general-purpose applications. Yet there are solutions to this problem, and one successful technique employed in the ADEPT system is that of "shared memory". Shared memory is achieved by using the basic mechanism for managing reentrancy, namely the program environment page map. Through the ADEPT SHARE Page call, an object program can request that designated pages of another program

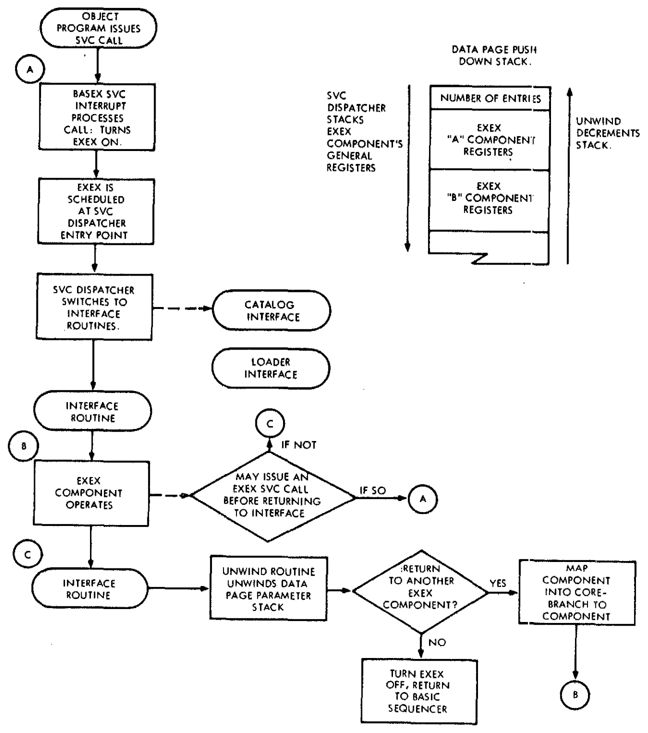

OBJECT PROGRAM ISSUES SVC CALL
A
BASEX SVC INTERRUPT PROCESSES CALL: TURNS EXEX ON.
EXEX IS SCHEDULED AT SVC DISPATCHER ENTRY POINT
SVC DISPATCHER SWITCHES TO INTERFACE ROUTINES.
CATALOG INTERFACE
LOADER INTERFACE
INTERFACE ROUTINE
B
EXEX COMPONENT OPERATES
C
IF NOT
MAY ISSUE AN EXEX SVC CALL BEFORE RETURNING TO INTERFACE
IF SO
A
C
INTERFACE ROUTINE
UNWIND ROUTINE UNWINDS DATA PAGE PARAMETER STACK
RETURN TO ANOTHER EXEX COMPONENT?
YES
MAP COMPONENT INTO CORE-BRANCH TO COMPONENT
NO
TURN EXEX OFF, RETURN TO BASIC SEQUENCER
B
DATA PAGE PUSH DOWN STACK.
SVC DISPATCHER STACKS EXEX COMPONENT'S GENERAL REGISTERS
NUMBER OF ENTRIES
EXEX "A" COMPONENT REGISTERS
EXEX "B" COMPONENT REGISTERS
UNWIND DECREMENTS STACK.

Figure 5—Block diagram of EXEX behavior and control

in the job be added to its map. If core page numbers are passed as parameters in various service calls, whole pages of data may be passed between programs. EXEX and many object programs operating under this system use this method for inter-program communication.

ADEPT operating on the IBM 360/50H restricts its user programs to 46 active core pages. However, by utilizing the GETPAGE call, an object program may acquire up to 128 drum pages and may subsequently activate and deactivate various page sets by utilizing another service call, ACTDEACT (activate/ deactivate). This scheme permits bulk data from disc storage to be placed on drum and operated upon at "swap" speeds. Thus skilled system users can achieve efficient use of time and memory by managing their own "paging". We consider this the best alternative considering the questionable state of other, automatic paging algorithms.[10,11,12,13] Most EXEX components use these calls for just such purposes. For example, the interface routines mentioned above use activate calls to "turn on" called components of the EXEX.

The Allocator component of ADEPT manages the page map for each program. This software map reflects the correspondence between drum and core pages, established initially by the SERVIS (service) component at load time. The Allocator's function is to inventory available core and drum pages by maintaining two resident system tables: one for core, the other for drum. Whenever drum pages are released or obtained, the Allocator updates the page map in the job's environment page. The Allocator processes the SHARE (page), GETPAGE, FREEPAGE, and ACTDEACT calls from EXEX and object programs. SERVIS allows a program at run time to add data pages or to overlay program segments from disc or tape. In so doing, SERVIS makes use of the various Allocator calls.

### Simulating console commands

An important attribute of ADEPT time-sharing is that nearly all the functions and services that can be initiated at the user's console can also be called forth within a user's program. A program designer can, for example, build a system of programs, which can operate in batch mode under the control of a program by issuing internal commands in much the same manner as the user sitting at the console. With this approach, the ADEPT batch monitor controls background tasks by simulating user terminal requests. Batch requests can be enqueued by users from any console and then processed in turn by this supervisor function.

### Armed interrupts and rescue function

The basic design of ADEPT conveniently provides for processing object program "armed" interrupt calls. This means that an object program is able to conditionally start (wakeup) and stop (sleep) the execution of its own programs, and others as well. The conditions for employing wakeup calls include too much elapsed time, or the occurrence of unpredictable but anticipated events, e g., errors and other program calls. In "arming" these "software-interrupt" conditions by object program calls, the program entry point(s) for the various conditions are specified. When such conditions occur, the operating system transfers to the specified entry point and gives the appropriate condition code. (Note that if we take this call one step further, and permit one object program to arm the software and hardware interrupts of another object program, we have the basic control mechanism necessary to permit the operation of "object systems.
necessary to permit the operation of "object systems,"
i.e., subexecutives—another level in the "onion skin" of ADEPT control.)

User programs interface with the ADEPT system primarily via the supervisor call (SVC) instruction; a secondary interface is provided via the program check interrupt that protects the program and system after various error conditions. The executive design allows user programs to trap all such interfaces with the system via its rescue arming mechanism. This means that one program can trap and get first-level control of all occurrences of SVC's and program checks within a single job. This mechanism also means, then, that the responsibility and meaning for these interfaces can be redefined at the user program level.

As of this writing, this mechanism is being employed to construct object systems for an improved batch monitor, an interface for the proposed ARPA Network,[14] and to experiment with automatic translators for compatibility with other operating systems. Other uses include improvements in program recovery in a variety of user tools, e.g., compiler diagnostics.

#### *Resource allocation, access, and management*

ADEPT system design, of course, includes a complete set of resource controls that monitor secondary storage devices.

## The cataloger

The Cataloger, an EXEX component, is functionally analogous to the core/drum Allocator, but is used for devices accessible by user programs. It maintains an inventory of all assignable storage devices, assigns unused storage on the devices, maintains descriptions of the files placed on these devices, controls access to these files, and—upon authorized request—deletes any file. Specifically, the Cataloger:

- Assigns storage on 2302, 2311 and 2314 discs.
- Assigns tape drives.
- Locates an inventoried file by its name and certain qualifiers that uniquely identify the file.
- Issues tape or disc pack mounting instructions to the operator when necessary.
- Verifies the mounting of labeled volumes.
- Passes descriptive information to the user program opening a file.
- Allows the user of a file to request more storage for the file.
- Denies unauthorized users access to files.
- Returns assigned storage to available storage whenever a file is deleted.
- Maintains a table of contents on each disc volume.

As the largest single component of the ADEPT Eexecutive (65,000 bytes), the Cataloger was written in a new, experimental programming language called MOL-360 (Machine-Oriented Language for the 360).[15] It is a "higher-level machine language" developed under an ARPA-sponsored SDC research project on metacompilers. It resolved the dilemma involving our desire for higher-level source language and our need to achieve flexibility with machine code. The Cataloger design and checkout, enhanced by the use of MOL-360, showed simultaneously the validity of MOL compilers for difficult machine-dependent programming.

## The SPAM component

SPAM is a BASEX component that permits symbolic, user-oriented I/O. It can be viewed as a special-purpose compiler that compiles symbolic user program I/O calls into 360 channel programs, and delivers them to the Input/Output Supervisor (IOS) for execution via the EXCP (execute channel program) call. The results of EXCP for the call are "interpreted" by SPAM and returned to the user program as status information. As such, SPAM represents a more symbolic I/O capability than the EXCP level. It provides a relatively simple method for executing the operations of reading, writing, altering, searching for, and positioning records within ADEPT cataloged and controlled disc-based and tape-based file structures.

## Resource management

As of this writing, the computer operator has a set of commands at his disposal that allow him to control the system resources. Various privileged on-line commands enable him to monitor the terminal activities of system users and to control assignment and availability of storage devices. However, there is an increasing need for a "manager" to be given more latitude in dynamically controlling the system resources and observing the status of system users, particularly because ADEPT was designed to handle sensitive information in classified government and military facilities. To meet these objectives, a design effort is under way that gives the computer operator system-manager status, with the ability to observe and control the actions of system users. The result will be a program that encompasses some of the management techniques reported by Linde and Chaney[16] tailored to present needs.

### *Swapping and scheduling user programs*

Most of the programs that run under ADEPT occupy all of the core memory that is not used by the resident Basic Executive (46 pages on the 360/50H). If the set of needed pages could be reduced considerable reduction in swap overhead could be expected. One way to achieve this is to mark for swap-out only those pages that were changed during program execution. The hardware needed to automatically mark changed pages is unavailable for the 360/50; however, through use of the store-protect feature on the Model 50, ADEPT software can simulate the effect and produce noteworthy savings in swap time.

## Page marking

Whenever a user program is swapped into core, its pages are set in a read-only condition. As the program executes, it periodically attempts to store data (write) in its write-protected pages. The resulting interrupt is fielded by the system. After satisfying itself that the store is legal for the program, the executive marks the target page as "written," turns off write-protect for that page, and resumes the program's execution. The situation repeats for each additional page written. At the completion of the program's time slice, the swapper has a map of all the program pages that were changed (implied in the storage keys with no write protection). Only the changed pages are swapped out of core. Measurement of this scheme shows that about 20 percent of the pages are changed; hence, for every five pages swapped in, only one need be swapped out, for a total swap of six pages, rather than the full swap of ten pages (five in, five out). The scheme makes the drum appear to be 40 percent faster.

The use of the storage protection keys is based on the functional status of each page rather than on some user identity. User programs always run with a program status word key of one, and the bits in the storage key associated with the programs start out at zero. After a page has been initially changed, its key is set to one also. The other bits in the key are used to indicate: first, a page is transient, not yet completely moved to or from swap storage; second, a page is unavailable, i.e., it belongs to someone else; third, a page is locked and cannot be swapped or changed; and finally, a page is fetch-protected because it may contain sensitive information.

## Scheduling algorithm

The scheduling algorithm provides for three levels of scheduling. Jobs that are in a "terminal I/O complete" state get first preference in the schedule. Jobs in the second level, or background queue, are run if there are no level-one jobs to run. A job is placed in level two when the two-second quantum clock alarm terminates its operation two consecutive times. Compute and I/O-bound programs are treated alike. A level-two job—when allowed to run—is given quantum interval equal to the basic quantum time multiplied by the scheduling level (i.e., 2 sec × 2 = 4 sec). However, a level-two background job may be preempted after two seconds for terminal I/O. Any operation a level-two job makes that terminates its quantum prematurely will return the job to a level-one status. The batch monitor job is run when the first two queues are empty. User programs may be written to overlap execution and I/O activity. Our choice of scheduling parameters for quantum size, and number of service levels was selected empirically and as a result of prior experience.[17]

A command SKED, which is limited to the operator's terminal, has the effect of forcing top priority for a job (the job stays at level one all the time). Only one job may run in this privileged scheduling state at a time.

*Pervasive security controls*

Integrated throughout the ADEPT executive are software controls for safeguarding security-sensitive information. The conceptual framework is based upon four "security objects": user, terminal, file, and job. Each of these security objects is formally identified in the system and is also described by a security profile triplet: Authority (e.g., TOP SECRET, SECRET), Need-to-Know Franchise, and Special Category (e.g., EYES ONLY, CRYPTO). At system initialization time, user and terminal security profiles are established by security officers via the system component SYSLOG. SYSLOG also permits the association of up to 64 passwords with each user. At LOGIN time, a user identifies himself by his unique name, up to 12 characters, and enters his private password to authenticate his identity. The LOGIN component of ADEPT validates the user and dynamically derives the security profile for the user's job as a complex function of the user and terminal security profiles. The job security profile is used subsequently as a set of "keys," used when access is made to ADEPT files. The file security profile is the "lock" and is under control of the file subsystem.

File access Need-to-Know is permitted for Private, Semi-Private, and Public use. With the CREATE command, a list of authorized users and the extent of their access authorization (i.e., read-only, write-only, read and write) can be established easily for Semi-Private files. Newly created files are automatically classified with the job's "high water mark" security triplet—a cumulative security profile history of the security of files referenced by the job. Through judicious use of the CHANGE command, these properties may be altered by the owner of the file.

Security controls are also involved in the control of classified memory residue. Software and hardware memory protection is extensively used. Software memory protection is achieved by interpretive, legality checking of memory bounds for I/O buffer transfers, legality checking of device addresses for unauthorized hardware access, and checks of other user program attempts to seduce the operating system into violating security controls.

The hardware protection keys are used to fetch-protect all address space outside the user program and data area. Also, newly allocated space to user programs is zeroed out to avoid classified memory residue.

for that page, and resumes the program's execution. The situation repeats for each additional page written. At the completion of the program's time slice, the swapper has a map of all the program pages that were changed (implied in the storage keys with no write protection). Only the changed pages are swapped out of core. Measurement of this scheme shows that about 20 percent of the pages are changed; hence, for every five pages swapped in, only one need be swapped out, for a total swap of six pages, rather than the full swap of ten pages (five in, five out). The scheme makes the drum appear to be 40 percent faster.

The use of the storage protection keys is based on the functional status of each page rather than on some user identity. User programs always run with a program status word key of one, and the bits in the storage key associated with the programs start out at zero. After a page has been initially changed, its key is set to one also. The other bits in the key are used to indicate: first, a page is transient, not yet completely moved to or from swap storage; second, a page is unavailable, i.e., it belongs to someone else; third, a page is locked and cannot be swapped or changed; and finally, a page is fetch-protected because it may contain sensitive information.

## Scheduling algorithm

The scheduling algorithm provides for three levels of scheduling. Jobs that are in a "terminal I/0 complete" state get first preference in the schedule. Jobs in the second level, or background queue, are run if there are no level-one jobs to run. A job is placed in level two when the two-second quantum clock alarm terminates its operation two consecutive times. Compute and I/O-bound programs are treated alike. A level-two job—when allowed to run—is given quantum interval equal to the basic quantum time multiplied by the scheduling level (i.e., 2 sec $\times$ 2 = 4 sec). However, a level-two background job may be preempted after two seconds for terminal I/O. Any operation a level-two job makes that terminates its quantum prematurely will return the job to a level-one status. The batch monitor job is run when the first two queues are empty. User programs may be written to overlap execution and I/O activity. Our choice of scheduling parameters for quantum size, and number of service levels was selected empirically and as a result of prior experience.[17]

A command SKED, which is limited to the operator's terminal, has the effect of forcing top priority for a job (the job stays at level one all the time). Only one job may run in this privileged scheduling state at a time.

*Pervasive security controls*

Integrated throughout the ADEPT executive are software controls for safeguarding security-sensitive information. The conceptual framework is based upon four "security objects": user, terminal, file, and job. Each of these security objects is formally identified in the system and is also described by a security profile triplet: Authority (e.g., TOP SECRET, SECRET), Need-to-Know Franchise, and Special Category (e.g., EYES ONLY, CRYPTO). At system initialization time, user and terminal security profiles are established by security officers via the system component SYSLOG. SYSLOG also permits the association of up to 64 passwords with each user. At LOGIN time, a user identifies himself by his unique name, up to 12 characters, and enters his private password to authenticate his identity. The LOGIN component of ADEPT validates the user and dynamically derives the security profile for the user's job as a complex function of the user and terminal security profiles. The job security profile is used subsequently as a set of "keys," used when access is made to ADEPT files. The file security profile is the "lock" and is under control of the file subsystem.

File access Need-to-Know is permitted for Private, Semi-Private, and Public use. With the CREATE command, a list of authorized users and the extent of their access authorization (i.e., read-only, write-only, read and write) can be established easily for Semi-Private files. Newly created files are automatically classified with the job's "high water mark" security triplet—a cumulative security profile history of the security of files referenced by the job. Through judicious use of the CHANGE command, these properties may be altered by the owner of the file.

Security controls are also involved in the control of classified memory residue. Software and hardware memory protection is extensively used. Software memory protection is achieved by interpretive, legality checking of memory bounds for I/O buffer transfers, legality checking of device addresses for unauthorized hardware access, and checks of other user program attempts to seduce the operating system into violating security controls.

The hardware protection keys are used to fetch-protect all address space outside the user program and data area. Also, newly allocated space to user programs is zeroed out to avoid classified memory residue.

Typically, the complete system reaches "on the air" status in less than a minute.

### *System instrumentation*

Many of the parameters built into the scheduling and swapping of early ADEPT versions were based upon empirical knowledge. The latest versions of the Basic and Extended Executives include routines to record system performance, reliability, and security locks.

Built into the BASEX is a routine to measure the overall and the detailed system performance.[20] Such factors as the number of users, file usage, hardware and software errors, and page transaction response time are recorded on unused portions of the 2303 drum. These measurements provide a better understanding of the system under a variety of inputs and give the designers insight into how the hardware and software components of the system affect the performance of the human user.

An AUDIT program was made part of the EXEX to record the security interaction of terminals, users, and files. AUDIT records EXEX activity in the areas of LOGIN, LOGOUT, and File Manipulation. This routine strengthens the security safeguards of the executive. Specific items that are recorded involve: type of event, user identification, user account number, job security, device identification, time of event, file identification, file security and event success. In addition, this routine provides accounting information and is used as a means of debugging the security locks of new system releases.

In addition to the BASEX recording function, several object programs have been written that simulate various modes of user activity and provide controlled job distributions. These programs, called "benchmarks," run under controlled conditions and enhance the means of improving system performance and throughput, as described elsewhere by Karush.[21] The programs are designed to gather performance measures on the major routines of the executive and have been of considerable help in system "tuning," because they reflect the effect of coding and design changes to various system routines. The routines in the executive that are of primary concern are the swapper, the scheduer, the terminal read/write package, and the interrupt handling processes. Attempts are being made to design a set of benchmarks that represent a typical job mix. However, we are primarily interested in measuring the performance of our system against various modifications of itself and in measuring its behavior with respect to different job mixes.

## SUMMARY

The ADEPT executive is a second-generation, general-purpose, time-sharing system designed for IBM 360 computers. Unlike the monolithic systems of the past,[1,2] it is structured in modular fashion, employing distributed executive design techniques that have permitted evolutionary development. This design has not only produced a flexible executive system but has given the user the same facilities used by the executive for controlling the behavior of his programs. ADEPT's security aspects are unique in the industry, and the testing and fabrication methods employ a number of novel approaches to system checkout that contribute to its operational reliability.

It is important to note that this system deals particularly well with size limitation problems of very large files and very large programs. The provisions made for multiple programs per job, active/inactive page status for programs larger than core size, page sharing between programs, common file access across programs within jobs, and the commitment of considerable space to active file environment tables (up to four pages worth) contribute to this success. Nevertheless, all these capabilities are designed to handle the smaller entities as well. We feel ADEPT-50 is a significant contribution to the technology of general-purpose time-sharing.

## ACKNOWLEDGMENTS

We would like to express our appreciation for the dedicated efforts of some very *adept* individuals who participated in the design and building of this time-sharing system. Our thanks go to Mr. Salvador Aranda, Mr. Peter Baker, Mrs. Martha Bleier, Mr. Arnold Karush, Mrs. Patricia Kribs, Mr. Reginald Martin, Mr. Alexander Tschekaloff and all the others who have followed their lead.

## REFERENCES

1 P CRISMAN editor
*The compatible time-sharing system: A programmer's guide*
MIT Press Cambridge Mass 1965

2 J SCHWARTZ et al
*A general-purpose time-sharing system*
Proc SJCC Vol 25 1964 397-411 Spartan Books Baltimore

3 E W FRANKS
*A data management system for time-shared file-processing using a cross-index file and self-defining entries*
AFIPS Proc Vol 28 1966 79-86 Also available as SDC document SP-2248 21 April 1966

4 R E BLEIER
*Treating hierarchical data structures in the SDC time-shared data management system (TDMS)*
Proc 22nd Nat ACM Conf Thompson Book Co 1967 41-49

5 E W DIJKSTRA
*The structure of T.H.E. multi-programming system*
C A C M Vol 11 No 5 May 1968

6 F J CORBATO V A VYSSOTSKY
*Introduction and overview of the multics system*
Proc FJCC Nov 30 1965 Las Vegas Nevada

7 B W LAMPSON
*Time-sharing system reference manual*
Working Doc Univ of Calif Doc No 30.1030
Sept 1965 Dec 1965

8 B W LAMPSON
*A scheduling philosophy for multi-processing systems*
C A C M Vol 11 No 5 May 1968

9 J H SALTZER
*Traffic control in a multiplexed computer system*
MAC-TR-30 thesis MIT Press July 1966

10 G H FINE et al
*Dynamic program behavior under paging*
Proc ACM 1966 223-228 Thompson Book Co Wash D C

11 E G COFFMAN L C VARIAN
*Further experimental data on the behavior of programs in a paging environment*
C A C M Vol 11 No 7 July 1968 471-474

12 L A BELADY
*A study of replacement algorithms for a virtual storage computer*
IBM Systems Journal Vol 5 No 2 1966

13 R W O'NEIL
*Experience using a time-shared multi-programing system with dynamic address relocation hardware*
Proc SJCC 1967 Vol 30 611-627 Thompson Book Co Washington D C

14 L G ROBERTS
*Multiple computer networks and intercomputer networks and intercomputer communication*
ACM Symposium on Operating System Principles
Oct 1-4 1967 Gatlinburg Tenn

15 E BOOK D C SCHORRE S J SHERMAN
*Users manual for MOL-360*
SCC Doc TM-3086/003/01

16 R R LINDE P E CHANEY
*Operational management of time-sharing systems*
Proc ACM 1966 149-159

17 P V McISSAC
*Job descriptions and scheduling in the SDC Q-32 time-sharing system*
SDC Doc TM-2996 June 1966 28

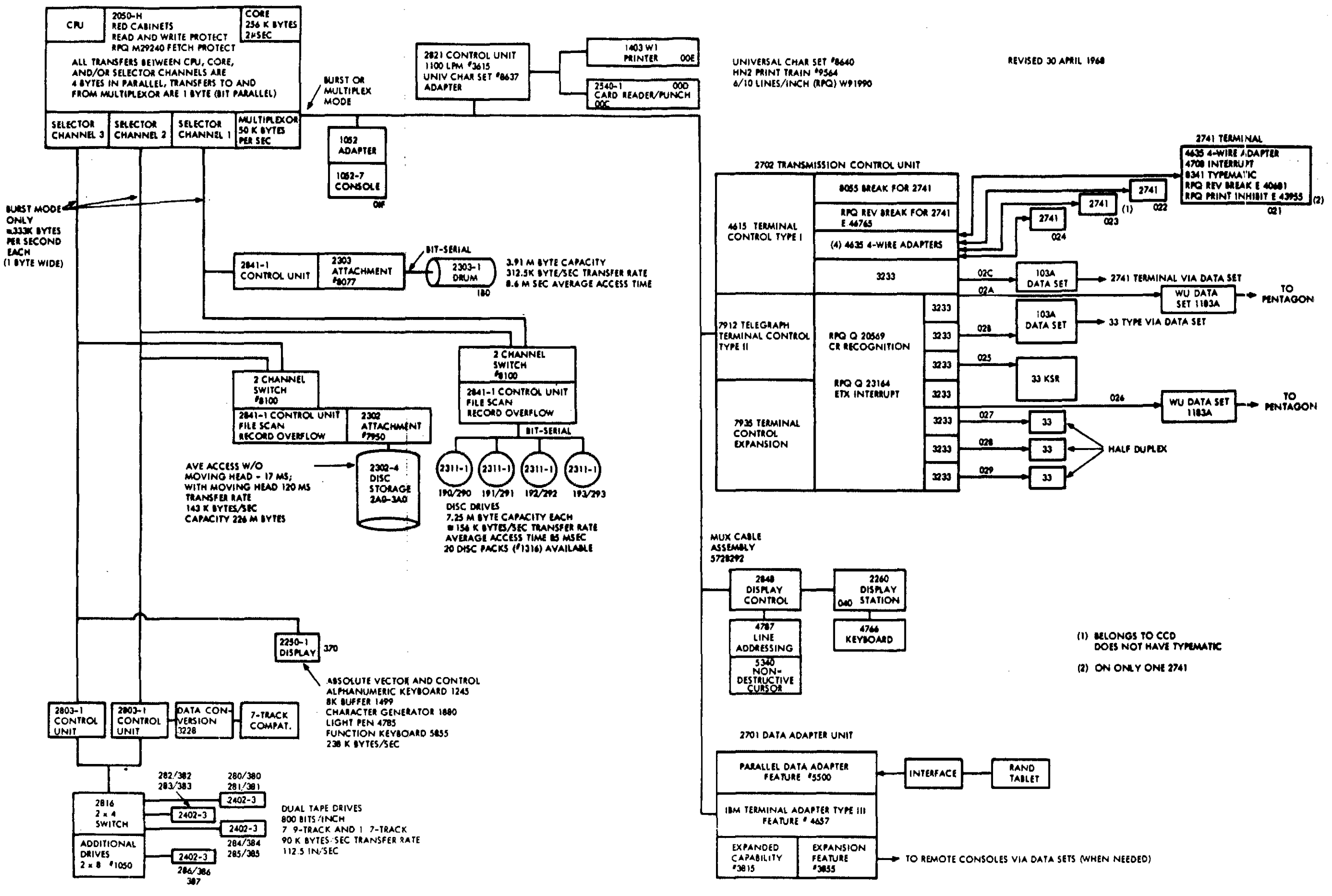
18 C WEISSMAN
*Security controls in the ADEPT-50 time-sharing system*
AFIPS Proc FJCC Vol 35 1969

19 W A BERNSTEIN J T OWENS
*Debugging in a time-sharing environment*
AFIPS Proc FJCC Vol 33 1968 7-14

20 A D KARUSH
*The computer system recording utility: application and theory*
SDC Doc SP-3303 Feb 1969

21 A D KARUSH
*Benchmark analysis of time-sharing system*
SDC Doc SP-3343 April 1969

## APPENDIX A: Advanced development prototype system block diagram.



# An operational memory share supervisor providing multi-task processing within a single partition

*by* J. E. BRAUN

*Penna. -N. J. -Md. Interconnection*
Philadelphia, Pa.

and

A. GARTENHAUS

*Applied Programming Services, Inc*
Philadelphia, Pa.

## INTRODUCTION

The real-time digital process control system, of which the Partition Share Supervisor is an operational feature, was designed and implemented to assist in the functions of monitoring, evaluating and controlling an interconnected system of electrical power utility companies. The main processing unit is located at the central control office with teleprocessing communications to remote lower level control centers.

The basic addressable unit within the main processor is the byte (8 data bits + 1 parity bit), with a word consisting of four bytes. There is a storage protect option which is implemented through assignment of storage and "keys" to contiguous 2048 byte blocks of memory. A group of memory blocks with matching protect keys comprise a partition or task area. This protection feature permits non destructive read-out across partition boundaries but will cause termination of any task which attempts to write in another task's memory area.

The arithmetic-logic unit maintains its current status in a program status word which contains such information as whether or not I/O is currently being permitted on each of the data channels, the protect key for the instruction presently being executed, present machine status, length of current instruction, the address of the next instruction to be fetched, etc. There are certain instructions within the instruction set which can only be executed when the machine is in the "supervisor" state, i.e., when the portion of the program status word which indicates machine status is correctly set. These instructions are classified as "privileged" instructions and perform such functions as disabling data channel interrupts, altering storage keys, resetting the program status word, etc.

The ability of the computer to disallow certain of its instructions when operating in the normal problem program state prevents inadvertent destruction of critical storage area or catastrophic conditions being caused by problem programs which could lead to system shutdown.

This system utilizes the independent I/O channel concept which permits the main processor to continue execution of program instructions while the channel transfers data from I/O devices into main storage by cycle interleaving.

The multi-tasking capability of the manufacturer supplied software support system permits priority scheduling of several tasks all utilizing the resources of one processing unit. The design of the real-time control system requires that it perform certain of its functions in a cyclic basis. Therefore, the internal storage has been divided into four task areas (partitions) with time dependent and critical programs placed in partitions with relatively higher priorities. The following task descriptions are listed in order of task priorities:

### Task 1 (core requirement) = 42K)

Task 1 is dedicated to the manufacturer supplied operating system (O/S) which contains supervisory routines, data management routines priority scheduler, etc.

### Task 2 (core requirement = 72K)

Task 2 incorporates the process control family of programs. It also includes the remote typewriter/card reader communications programs since they use little processing time and benefit from both the independence of input/output channel operations and quick response time available to the task. During power system emergency situations, Task 2 additionally initiates routines which, due to their critical nature, retain system resources and dispatch emergency communications until the disturbance is relieved.

### Task 3 (core requirement = 40K)

Task 3 contains special digital console message processing routines, text output generators for programs operational within Task 2, routines for processing card inputs from the telecommunications system and routines which monitor and control inter-task communications.

### Task 4 (core requirement = 6K)

Task 4 is the Partition Share Supervisor (PSS) which causes Tasks 5 and 6 to share the remaining available memory. The detailed description of this task is the subject of this paper.

### Task 5 (core requirement = 96K)

Task 5 consists primarily of scientific application programs. These programs are run as required either on special demand from real-time on-line tasks or periodically with the length of the period depending on the nature of the program.

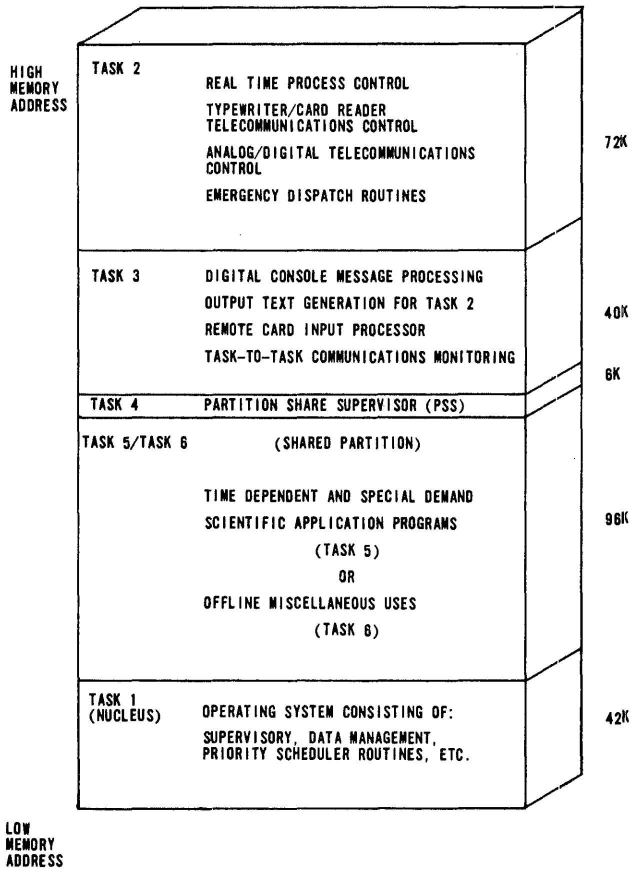

HIGH MEMORY ADDRESS
TASK 2
REAL TIME PROCESS CONTROL
TYPEWRITER/CARD READER TELECOMMUNICATIONS CONTROL
ANALOG/DIGITAL TELECOMMUNICATIONS CONTROL
EMERGENCY DISPATCH ROUTINES
72K
TASK 3
DIGITAL CONSOLE MESSAGE PROCESSING
OUTPUT TEXT GENERATION FOR TASK 2
REMOTE CARD INPUT PROCESSOR
TASK-TO-TASK COMMUNICATIONS MONITORING
40K
6K
TASK 4
PARTITION SHARE SUPERVISOR (PSS)
TASK 5/TASK 6
(SHARED PARTITION)
TIME DEPENDENT AND SPECIAL DEMAND SCIENTIFIC APPLICATION PROGRAMS (TASK 5)
OR
OFFLINE MISCELLANEOUS USES (TASK 6)
96K
TASK 1 (NUCLEUS)
OPERATING SYSTEM CONSISTING OF:
SUPERVISORY, DATA MANAGEMENT, PRIORITY SCHEDULER ROUTINES, ETC.
42K
LOW MEMORY ADDRESS

Figure 1—Initial memory configuration with task functional descriptions and relative locations shown

### Task 6 (core requirement = 96K)

This task is the off-line* task and is dedicated for miscellaneous uses such as compiles, assemblies, accounting routines, etc.

Figure 1 is a functional diagram of the tasks just discussed and shows their relative locations in computer memory.

*General discussion*

### Task dispatching

Task dispatching is under the control of the operating system. From a conceptual standpoint, the operating system can be considered to be the only main program in storage and all other tasks within the computer as subroutines.

* The term off-line is used in this paper when referring to tasks which do not directly operate within the real-time environment. This use is similar to the term "background" which the reader may have previously encountered.

The dispatching function consists of allocating the resources of the processor to the highest priority task which is in the "ready" state. When no tasks are in the ready state, the processor is not working and is in a wait state. When any task reaches a point where it no longer can process until the completion of some event (such as an I/O operation), it relinquishes control of computer facilities to lower priority tasks via the scheduler. It will regain these facilities when the event it is awaiting is completed and there are no higher priority tasks which are in the ready state.

## Inter partition communication

The subject real-time system requires that operational tasks be able to communicate for the purpose of exchanging information such as live data, requests to run various subtask routines, etc. Tasks which communicate with other tasks are equipped with intertask communication routines which are considered the highest priority routines within the individual task. In this fashion, when the task is dispatched, the internal task priority scheme allows the communication routines to be processed first. Furthermore, any task can be interrupted to allow its communication routines to operate. Thus tasks can communicate at any time (asynchronously).

## Partition sharing

The Partition Share Supervisor (PSS) is required to be able to handle three basic functions:

1. Suspend processing of the off-line task when required.
2. Load and process the lowest priority on-line task (LPOL).
3. Upon completion of (2) above, be able to restore and restart the off-line task.

There are two conditions under which PSS suspends off-line processing. One is when the previously set real-time clock causes an interrupt. This interrupt is recognized as indicating the LPOL is to be recycled for a periodic run. The other is when a communication is received from another task indicating that one of the routines within the LPOL task is to be executed.

Figure 1 shows the computer configuration in the normal mode. Normal mode is considered to be when the shared partition is occupied by off-line programs. Note that there are four problem program partitions (excluding the nucleus).

Figure 2 shows the configuration when the off-line programs are "rolled out" and the LPOL programs are operational. There are now three problem program

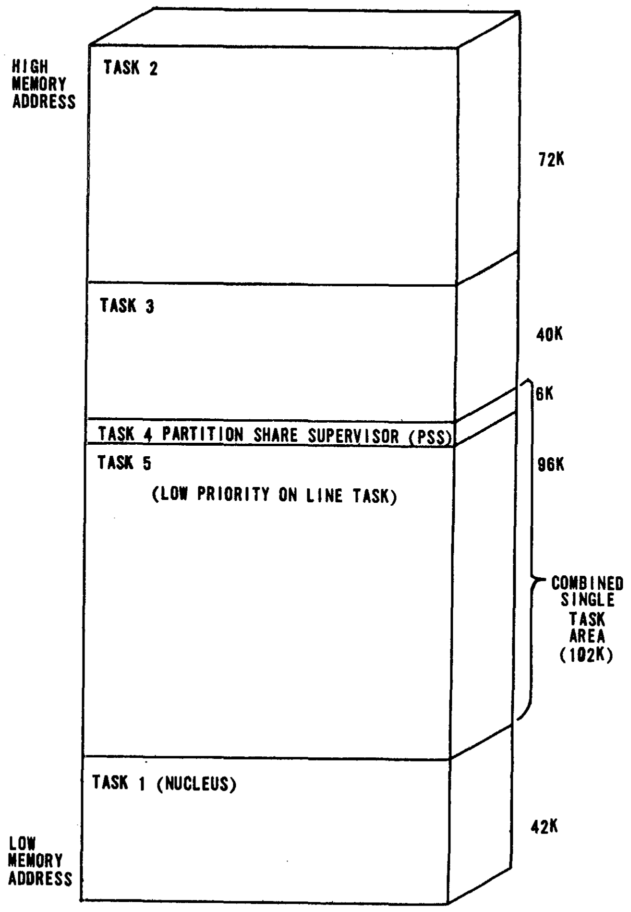

HIGH MEMORY ADDRESS
TASK 2
72K
TASK 3
40K
6K
TASK 4 PARTITION SHARE SUPERVISOR (PSS)
TASK 5
(LOW PRIORITY ON LINE TASK)
96K
COMBINED SINGLE TASK AREA (102K)
TASK 1 (NUCLEUS)
42K
LOW MEMORY ADDRESS

Figure 2—Showing memory configuration when low priority on line (LPOL) task is active

partitions and the area dedicated to the PSS and LPOL tasks is one contiguous partition.

### *Detailed discussion*

The following description details the operations involved in reconfigurating the system from that of Figure 1 to that of Figure 2 and returning to that of Figure 1.

As previously stated, the PSS task is initiated for one of two reasons:

1. Timer interrupt indicating a need to run the LPOL task for time dependent programs.
2. External interrupt triggered by communication from another task indicating a need to process a requested program.

Prior to either type of interrupt, the PSS task is in a wait state (i.e., the task cannot be dispatched until the completion of one of the above two events).

Upon being initiated, PSS takes the following steps:

1. Places its own task in the supervisor state in order to allow execution of privileged instructions required to modify system control blocks in the nucleus, override the storage protection feature, and disable system interrupts at critical times.
2. Allows all outstanding I/O to complete in the off-line partition (quiescing the partition).
3. Erases the boundary between the PSS task and off-line task.
4. Deletes reference to the now non-existent off-line task from operating system control blocks.
5. Writes a copy of the off-line partition, which is now an extension of the memory area of the PSS task, on a disc file.
6. Reads the LPOL task into the vacated area.
7. Executes the LPOL task.

At this point, we have gone from the configuration shown in Figure 1 to that of Figure 2 and the LPOL task is now able to process its requests. Upon completion by the LPOL task of all required processing, the following steps are taken by PSS to return to the off-line configuration:

8. Writes the LPOL task on a disc file.
9. Reads the off-line task into the vacated area.
10. Re-establishes task boundaries erased in 3.
11. Restores system reference to the off-line task.
12. Places the PSS task in a "wait state" awaiting an interrupt which will cause a recycle.

At this point, the off-line task is fully restored to the system and in a "ready state". It will then be redispatched by the task dispatching routines on a priority basis.

## System control blocks

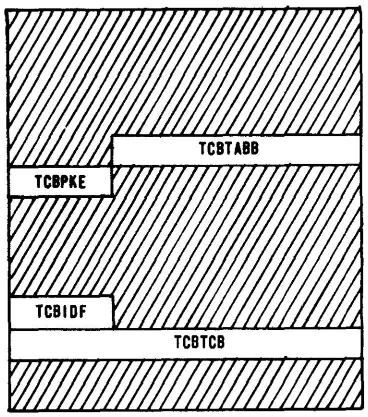
Prior to a detailed discussion of PSS mechanics, we will discuss relevant system control blocks utilized in effecting partition sharing.

### Task Control Block (TCB)

There is a TCB associated with each task. Contained in the TCB are various boundaries, indicators, etc., used in performing task control. Figure 3 shows those fields (with references labeled as used in this paper) which are accessed or modified by PSS.

### TCB List (TCBLIST)

The TCBLIST is located in the nucleus and is a list of TCB locations in order of task priority. There is an entry in the list for each task in the system (see Figure 4).



| FIELD | COMMENTS |
|---|---|
| TCBTABB | POINTER TO TASK MSS (BOUNDARY BOX-SEE FIG.5) |
| TCBPKE | CONTAINS STORAGE PROTECTION KEY FOR THE TASK |
| TCBIDF | TASK IDENTIFICATION NUMBER |
| TCBTCB | POINTER TO NEXT LOWER PRIORITY TASK TCB |

Figure 3—Task control block (TCB)

### Task Area Boundary Block (TABB)

There is a TABB associated with each task. The TABB contains addresses defining the upper and lower boundaries of the task region and also has a pointer to the first free area label within the task. The format of a TABB is shown in Figure 5.

### Free Area Label (FAL)

There is an FAL which is an integral part of every available free storage area in memory. An FAL is

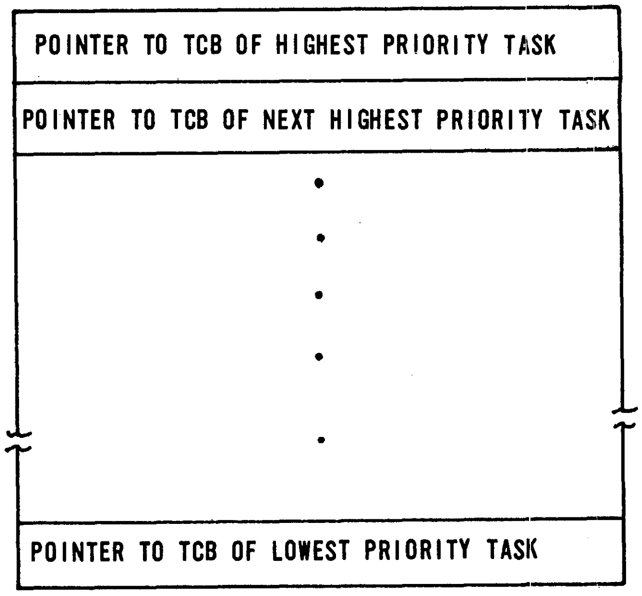
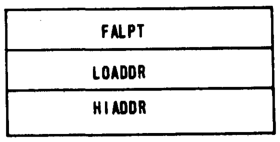

POINTER TO TCB OF HIGHEST PRIORITY TASK
POINTER TO TCB OF NEXT HIGHEST PRIORITY TASK
POINTER TO TCB OF LOWEST PRIORITY TASK

Figure 4—TCB list (TCBLIST)



| LABLE | COMMENT |
|---|---|
| FALPT | POINTER TO FIRST FREE AREA LABEL (FAL) WITHIN TASK AREA. (SEE FIGURE 6) |
| LOADDR | THE ADDRESS OF THE LOW BOUNDARY OF THE TASK |
| HIADDR | THE ADDRESS OF THE HIGH BOUNDARY OF THE TASK. |

Figure 5—Task area boundary block (TABB)

effectively a label for each free storage area which defines the size of it and contains a linkage pointer to the next FAL. The format of an FAL is shown in Figure 6.

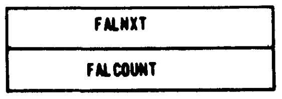
### Input/Output Request Element (IORE)

There is a chain of IOREs for all outstanding or queued I/O operation requests from any partition. Each IORE contains information used by the system I/O interrupt handling routines as I/O operations are completed. Figure 7 shows the format of an IORE.

### System Vector Table (SVT)

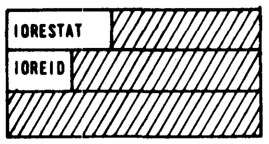
The SVT is resident in the nucleus and contains essential pointers required by the operating system. Included is a pointer to the start of the IORE chain. The location of the SVT is retrieved from a fixed memory location which is conditioned with the SVT address during system initialization.

As mentioned under General Discussion, the PSS task is required to run in supervisor state at times. Although the state of the PSS task changes from problem to supervisor and back throughout its execution, these changes of state will not be noted in this discussion. It should be understood that PSS operates in problem state at all times where it is not required to be executing privileged instructions, modifying storage in another partition or the nucleus, or disabling I/O interrupts.



| | |
|---|---|
| FALNXT | POINTER TO NEXT FAL IN THE CHAIN OF FAL'S.. NOTE: IF THIS FIELD IS ALL ZEROS, THIS IS THE LAST FAL IN THE CHAIN. |
| FALCOUNT | AMOUNT OF FREE MEMORY AVAILABLE STARTING AT THE BEGINNING OF THIS FAL. |

Figure 6—Free area label (FAL)



| | |
|---|---|
| IORESTAT | STATUS INDICATOR FOR THIS IORE. THE LAST IORE IN THE CHAIN HAS AN IORE STAT FIELD WITH A VALUE OF 1. |
| IOREID | FIELD SET TO SAME ID NUMBER AS THAT OF THE TCBIDF FIELD OF THE TASK WHICH INITIATED I/O REQUEST (SEE FIGURE 3) |

Figure 7—I/O request element (IORE)

## Quiescing a partition

Prior to rolling out the off-line partition, PSS must be sure all I/O is quiesced in order to prevent the I/O supervisor routines from accessing some storage area which is in a transitory state.

There is an IORE for all outstanding and queued I/O requests. Within each IORE is an identification number field (IOREID—see Figure 7) which links it with the initiating task. When that task is involved in an I/O operation, the TCBIDF field of the TCB (Figure 3) has a task identification number that will match the IOREID field of some active IORE.

As I/O interruptions occur, the I/O Interrupt Handler services the interrupt and removes the appropriate IORE from the chain and makes it inactive.

Partition quiescing is accomplished by initially disabling I/O interrupts, obtaining the TCBIDF field from the TCB of the task involved, locating the IORE chain by using the pointer in the SVT, and scanning the IOREs checking for IOREID fields which match the TCBIDF field of the TCB. If none are found, there are no IOREs for the task and it is already in a quiescent state. If any are found, then the task has a pending I/O interrupt or outstanding I/O requests. If this is the case, PSS enables interrupts allowing the I/O Supervisor to process, if necessary, and then immediately disables them. If the I/O in question has been completed, the IORE will have been removed from the chain during the time interrupts were enabled.

PSS restarts at the beginning of the chain and checks again, repeating the above steps until it comes to the end of the chain without having found any active elements for the task. When it reaches this point, there are no longer any IOREs associated with the task and it is in fact quiescent.

It should be noted that since the PSS task has a higher priority than the task to be quiesced, it does not allow any new I/O requests to be initiated by that task since PSS retains the computer resources.

## Erasing of a partition boundary and task deletion

There is control information which is received by the communications routines within the PSS task which must be accessible to the LPOL task for both reading and writing (such as indications which LPOL routine to is be run, the replacement value for the next cycle time which is calculated by the LPOL task as a function of its current running time, entry point addresses of routines mutually shared by the PSS and LPOL tasks, etc.). Additionally, task management is greatly facilitated by extending the PSS task area to include the LPOL function while controlling via the PSS Task Control Block (TCB) rather than modifying the off-line task TCB or creating a new one.

In order to make the shared task area a memory extension of the PSS task, the memory areas must be linked. This is achieved by modifying the TABB (see Figure 5) of the PSS task so that the LOADDR field points to the low address of the shared task. Figures 8 and 8a show the pointer relationships before and after these TABB modifications.

The storage protection feature must now be satisfied to make the two storage areas completely contiguous. Since there is a mismatch in storage keys between the PSS and shared tasks, the keys associated with each protected block of memory within the shared task are reset to match those of the PSS task. At this point, the two task areas have become a contiguous block of memory assigned to the PSS task area.

Figure 9 shows how TCBs are linked together within the system. Note that each entry in the TCBLIST points to a TCB and each TCB points to the next lowest priority TCB in the chain. Figure 9a shows the arrangement of the TCBLIST and the TCBTCB field in the next-to-last TCB in the chain after modification to three partitions. This has been done by replacing the pointer to the last TCB in the TCBLIST with a pointer to the next-to-last TCB, and setting TCBTCB field of the next-to-last TCB to zero. These modifi-

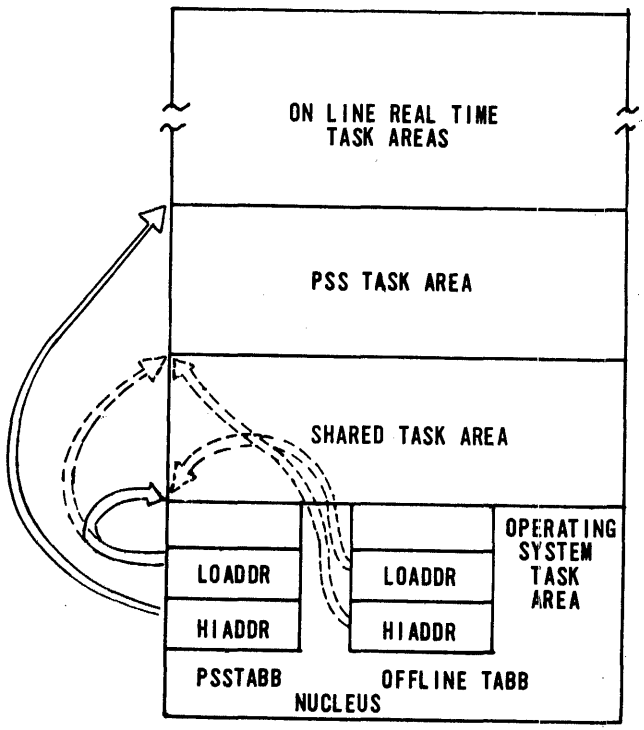


Figure 8a—TABB pointers after modification

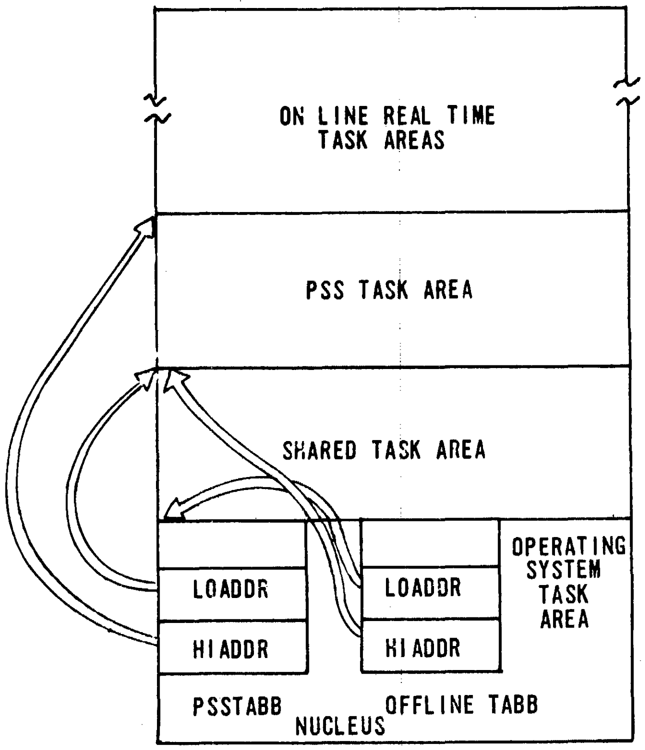


Figure 8—TABB pointers in PSS and offline task prior to modification

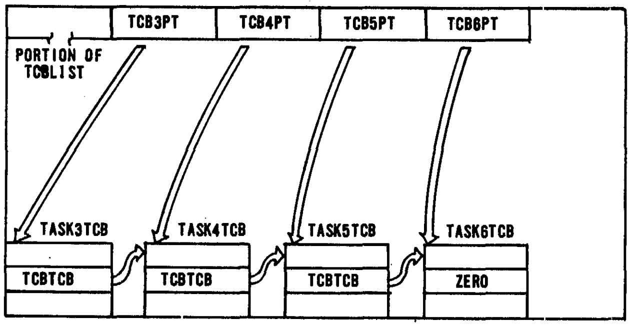


Figure 9—Portion of nucleus showing TCBLIST and TCBTCB pointer relationship prior to modification

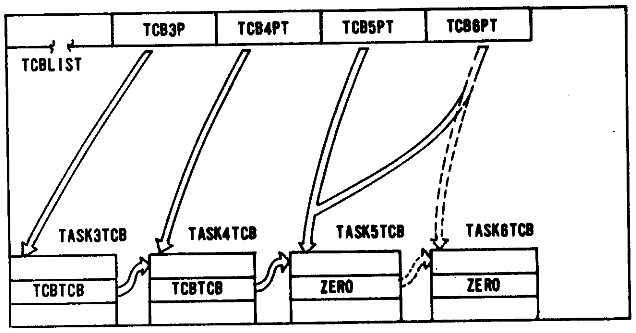


Figure 9a—TCBLIST and TCBTCB pointers after modification

cations have additionally made the last task non-existent to the operating system.

## Rollout/Rollin

The process of rolling out the off-line task and rolling in the LPOL task is a straightforward write/read operation to a disc file. Since storage is divided into 2048 byte units for assignment of storage keys, the task area read or written is some multiple of 2048 bytes in length. Thus the records are read or written in 2048 byte blocks for purposes of simplicity and efficiency.

## Free area modification

The PSS and LPOL tasks now occupy the same task area. It is necessary, therefore, to make certain modifications which will cause all requests for work storage to be satisfied from that portion of the task area wholly dedicated to the LPOL task. Although no task boundary exists between LPOL and PSS, if work storage were to be allocated from the PSS domain, it would not be subsequently saved and restored in future cycles since the PSS area is not included in the dynamic area which is stored on the disc file.

Figures 10 and 10a show how these modifications are accomplished. Initially (Figure 10) the FALPT field of the PSS TABB is pointing to the free area within what was its own task area. This is the normal condition for this pointer when there is an operating off-line task. However, we have modified the configuration to three task areas and we now wish to make the only available free area all exist in the LPOL area. Figure 10A shows that the FALPT field of the PSS TABB has been re-pointed to the first FAL within the LPOL task area.

At this point, the LPOL task is ready to process

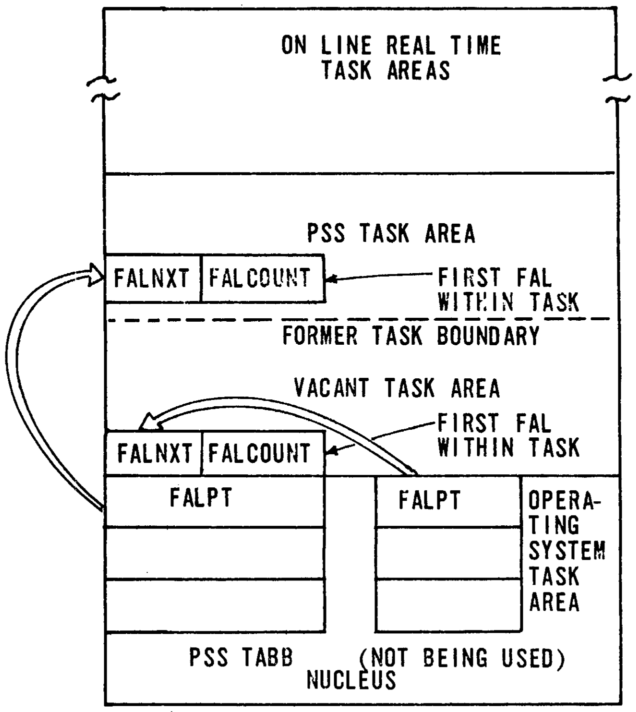


Figure 10—FALPT relationship with FAL locations prior to modification

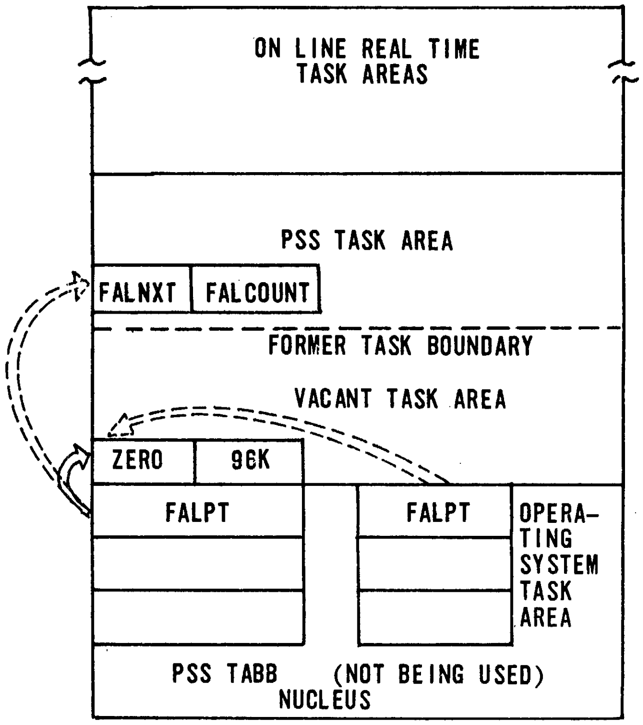


Figure 10a—FALPT fields after modification

whatever request caused it to be activated. We have now covered steps 1 through 7 under General Discussion. In returning from the three partition to the four partition environment, the steps are essentially the reverse of those detailed.

Upon restoring the off-line task, PSS enters a wait state and will be restarted as previously outlined. The task dispatcher portion of O/S will restart the off-line task as soon as there is available computer time and no higher priority tasks require the computer resources.

## Initialization

The initialization process for PSS consists of:

1. Suspending of off-line processing.
2. Reconfiguration from four to three partitions.
3. Rolling out the off-line task.
4. Making the off-line task area one contiguous free area.
5. Loading the LPOL task and allowing it to initialize itself.
6. Rolling out the LPOL task.
7. Rolling in and restarting the off-line task.
8. Entering the normal cycle at the wait point.

Step 4 above has not been previously covered in detail. In order to force the initial loading of LPOL into the desired location, the FALs for PSS are initially modified. Figures 10 and 10A show the PSS TABB before and after this is done. The FALPT field of the PSS TABB initially points to the first FAL within the PSS area. The FALPT field of the LPOL TABB points to the first FAL of its task area. By altering the FALPT of the PSS TABB to make it point to the LPOL first FAL and by altering the FAL by both making it the last FAL in the chain and indicating one large block of free memory, we have created a large free area available to PSS for loading the LPOL programs.

As the LPOL task acquires and releases memory blocks for work storage, the FALs within the area are modified by the operating system consistent with memory availability. PSS simply saves the pointer to the first LPOL FAL prior to each rollout and restores it after rollin and prior to reinitiating LPOL. Continuity of FAL linking is maintained in this fashion.

### *Special handling*

There are occasions when the off-line partition cannot be quiesced. This could be caused by a card reader jam, a printer being out of paper, etc., causing an IORE associated with the I/O to remain linked in the chain beyond some reasonable amount of time (presently 10 seconds). These conditions are relatively infrequent; however, provision has been made for them by advising the operator via the computer console typewriter and an attention bell that the off-line task is non-quiescent and requires attention.

The memory area actually required by PSS is less than 6K. However, in order to initially load PSS into memory, a large enough partition must be available to furnish the operating system job scheduler routines their required amount of core. This requirement is in the order of 24K. Thus there is a pre-initialization phase during which PSS changes the initial configuration (Figure 11) of 50K and 52K to 6K and 96K for the PSS and off-line tasks, respectively (Figure 1). The technique for doing this will not be detailed; however, the essential steps are as follows:

1. Referring to Figure 12, the initial PSS task area is shown in three segments (B, C, D) and the initial off-line task area is shown in one segment (A). The PSS Pre-Initializer is loaded by the operating system into area B.

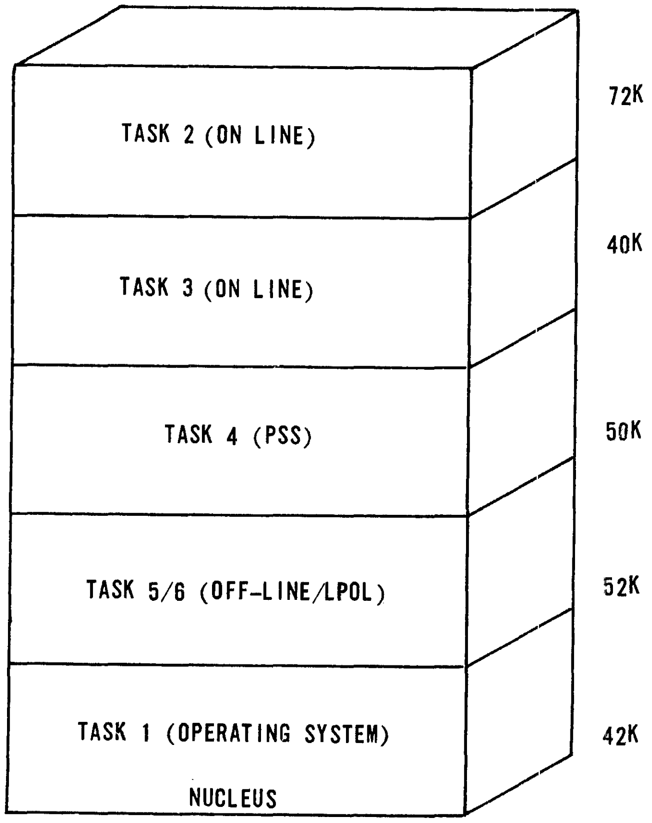


Figure 11—Initial task core allocations

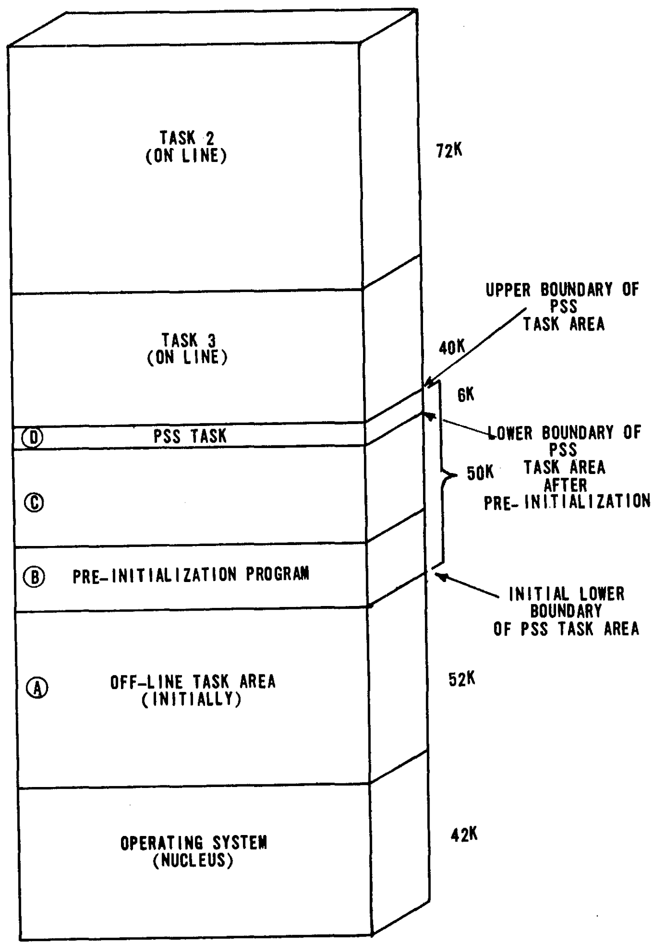

TASK 2
(ON LINE)
72K
TASK 3
(ON LINE)
40K
UPPER BOUNDARY OF PSS TASK AREA
6K
D PSS TASK
LOWER BOUNDARY OF PSS TASK AREA AFTER PRE-INITIALIZATION
50K
C
B PRE-INITIALIZATION PROGRAM
INITIAL LOWER BOUNDARY OF PSS TASK AREA
A OFF-LINE TASK AREA (INITIALLY)
52K
OPERATING SYSTEM (NUCLEUS)
42K

Figure 12

2. In order to place the PSS main program in the area where it can control storage, it must be forced into area D. To achieve this, the task area boundary block is modified to make area D free and areas B and C unavailable.
3. The PSS main program is loaded into area D.
4. The off-line boundary block is modified to include areas B and C as free areas.
5. Control is passed to PSS main.

The configuration is now that of Figure 1.

## CONCLUSION

Implementation of PSS has effectively added 96K of additional processor memory to the real-time system of which it is an integral part. This coupled with the facility to process off-line tasks while having an available stand-by on-line task, has greatly enhanced the capability of the system. The application of PSS has effected a maximal utilization of computer resources by the system.

## REFERENCES

1 *IBM System/360 operating system control blocks*
Form No C28-6628

2 *IBM system/360 operating system input/output supervisor*
Program Logic Manual Form No Y-28-6616

3 *IBM system/360 operating system control program with MFT*
Program Logic Manual Form No Y27-7128

4 *IBM system/360 operating system fixed task supervisor*
Program Logic Manual Form No Y28-6612

# Structured logic

*by* R. A. HENLE, I. T. HO, G. A. MALEY
and R. WAXMAN

*IBM Components Division*
Hopewell Junction, N.Y.

## INTRODUCTION

Large-scale integration for computer applications has been predicted for several years, but close examination shows that the progress has been uneven. Memory designers continually demand higher levels of integration for larger and faster memory systems, and new memory concepts are being developed to further exploit the characteristics of large-scale integration. The one-thousand-circuit chip will become nothing more than a milestone.

But what of the logic area? Here, we struggle along hoping to find some high-volume applications for chips with a mere fifty circuits. When we design a medium-sized machine we find that so much unit logic is required that the average level of integration falls below ten. Orderly memory and random logic integrated circuit fabrication procedures are growing so different that thought is being given to building different types of manufacturing facilities. This represents a rather drastic approach and in the authors' opinions may prove unnecessary.

The success to date in memory is encouraging, for it gives direction to logic. Memory products should therefore be examined critically for they may well hold the key to success for logic products. The salient features of a chip used in a memory product are:

- Regularity. Memory arrays are regular in components and wiring. The layout geometry is well defined and can be highly optimized for total chip utilization.
- Low Power. Memory systems are designed and partitioned so that all circuits on a chip do not dissipate maximum power at the same time.
- Well-Defined Function. The memory chip designer knows exactly how his chip fits into the entire memory system. He therefore can optimize on a high level. As examples, he uses special circuits for the latch functions and uses decoders redundantly to save pads.
- Volume. While the initial memory chip design is quite complex, the volume requirement makes the initial design cost nearly negligible. With this ground rule the chip can be highly engineered, and nearly order of magnitude improvement can be expected and obtained.

Structured logic, or array logic as it is sometimes called, is an attempt to design logic with more of the characteristics of memory. Many unsuccessful starts have taken place, but we shall discuss some of the more successful efforts. We shall also add some thoughts of our own, but it should be pointed out that the problem is far from solved.

### *Logic arrays*

The basis of all array logic is a matrix of elements with programmable interconnections. Diode structures have been proposed in the past, and a matrix of common collector transistors is of recent interest. The transistor array is programmed in the factory by connecting or not connecting the emitter of each transistor to a common line. (See Figure 1.) We shall use transistor arrays in our examples, for that is what we have been working with, but diode arrays should not be ruled out.

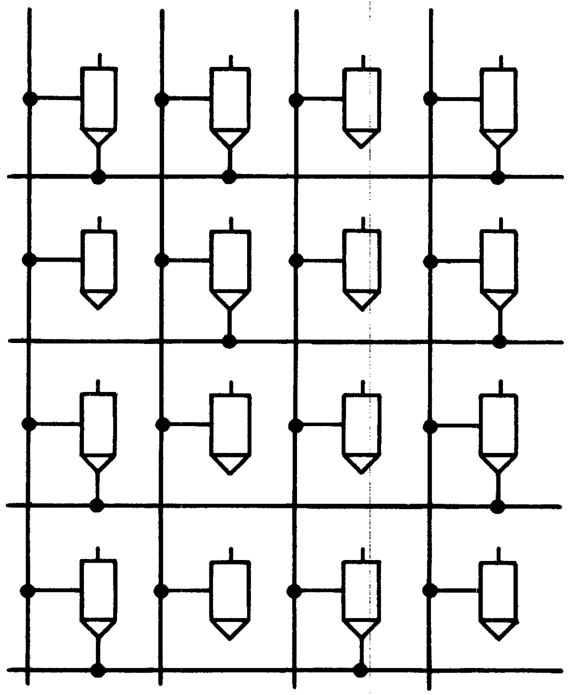


Figure 1—A transistor array

## The ROS

The read-only store (ROS) array in its simplest form uses two decoders to feed the array: one feeds the horizontal lines and the other the vertical lines, as shown in Figure 2. A particular grid position in the array is selected by activating the appropriate horizontal and vertical decoder lines. The addressed cell of the array is located at the intersection of the two activated lines. If the emitter at this address is connected to the horizontal decoder line, then a 1 has been programmed into this particular cell in the array. If the emitter is unconnected, a 0 is said to be programmed into the array. The presence of the programmed 1 or 0 is sensed at the output when that particular cell is addressed. The horizontal output lines are dot ORed together to produce one common output line, as shown in Figure 3.

Conceptually, the ROS is related directly to a Karnaugh map, one bit position in the array for each square in the appropriate Karnaugh map. Figure 4 depicts the four-variable K-map that relates to the ROS of Figure 2. This relationship proves the universality of a ROS, for any Boolean function that can be K-mapped can be implemented directly. Universality is the feature of the ROS chip most often described as an asset, but in practice it is seldom useful except in code translators. The Boolean functions used in the design of any computer are definitely not random and not evenly distributed among all possible functions of n variables. This fact is well documented in the many failures with other universal logic blocks (ULB's). The real problem with the ROS array is that it doubles in size each time an input variable is added. This doubling in size is necessary to maintain the dubious value of being universal.

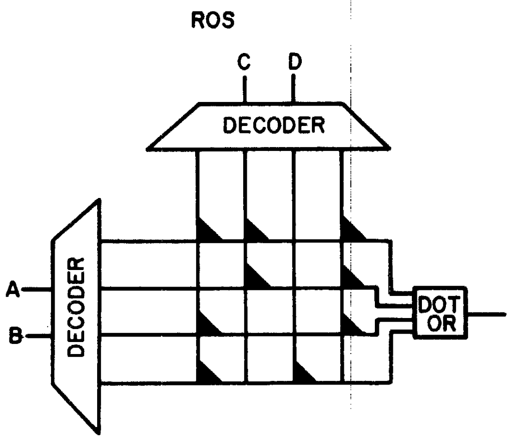


Figure 2—Read-only store

## The ROAM

The read-only associative memory (ROAM) is a

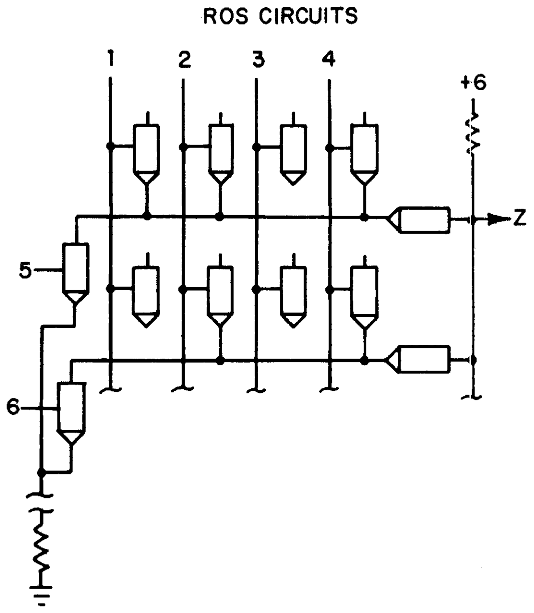


Figure 3—Read-only-store circuits

K-MAP

| A,B \ C,D | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | 1 | 0 | 1 |
| 01 | 0 | 1 | 0 | 1 |
| 11 | 1 | 0 | 0 | 1 |
| 10 | 1 | 0 | 1 | 0 |

Figure 4—Karnaugh map

matrix of common collector transistors that may be programmed by connecting or not connecting the base of each transistor to a common line in its own column (Figure 5.) The emitters of each row are commoned and feed the emitter of an output transistor. Each row of array transistors and the associated output transistor form a current switch.

Through phase splitters, each input variable has both true and complement lines available to the array. Hence, each variable controls a true line and a complement line (column) in the array. This gives rise to the word "associative" in the name. By programming each row in the array to a particular pattern of 1's and 0's, the input word pattern will "associate" (compare) with the appropiate row in the array. If there is no match, the outputs will remain logical zeros. If at least one row has a pattern the same as the input pattern, there will be a logical one output on that horizontal line (row).

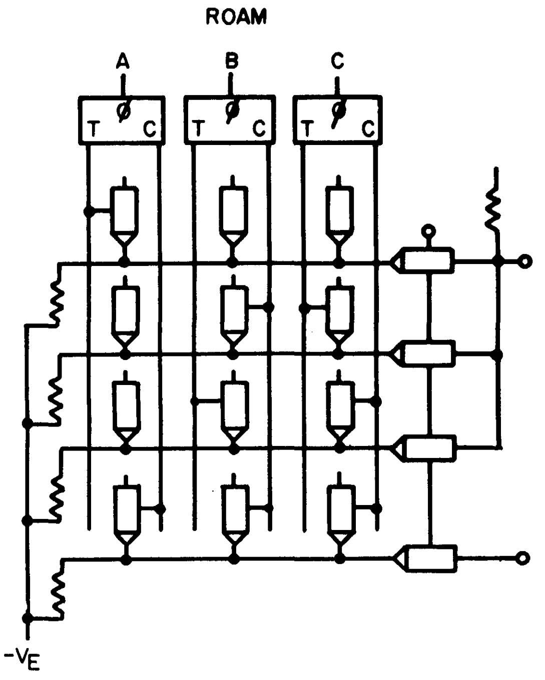


Figure 5—Read-only associative memory

To program the array, each base is tied to a true line (column), a complement line (column), or is left floating. Thus, for a base tied to a true line, a 1 on that input line will yield a 1 at the emitter and a 1 at the output, since the row of emitters effectively forms a DOT-OR (positive logic). Bases tied to a true line are equivalent to a logical 1, since a 1 at that input causes a 1 at the output.

Conversely, a base tied to a complement line is equivalent to a logical 0. A 0 at a particular input raises the complement line of the phase splitter, thereby raising to the 1 level all emitters of transistors in that column that have their bases tied to the complement line (column).

If the base is left floating, that array grid position is effectively a DON'T CARE. That is, the output line will not be raised to 1 by either a 1 or 0 at that transistor's column input.

Figure 6 illustrates the implementation of an adder position with SUM and CARRY outputs using a ROAM array. A black triangle connecting a vertical line and a horizontal line indicates a base connection; lack of a black triangle indicates a floating base. Note that if a true line is connected, then the complement line is not connected, and vice versa for each array grid position. Thus, at most, only 50 percent of the horizontal and vertical intersections will ever be used.

To conceptually understand the ROAM and relate it to the Karnaugh Map it is convenient to think in terms of negative logic. Thus, down levels are logical 1, the commoned emitters of each row form a DOT-AND (all emitters down results in a down level, any emitter up results in an up level), and dotting the output transistors results in a DOT-OR.

Each row of the ROAM represents a term of a logical expression in the sum-of-products form. The logical expression CARRY $= B \cdot C + A \cdot B + A \cdot C$ is in sum-of-products form, and $B \cdot C$, $A \cdot B$, and $A \cdot C$ are each terms of the expression. Each term may be implemented on one row of the ROAM. For example, Figure 6 illustrates the implementation of the CARRY function. Note that the A true and B true columns are both connected to a transistor base in the second row of the ROAM array, yielding the term $A \cdot B$. The three rows $B \cdot C$, $A \cdot B$, and $A \cdot C$

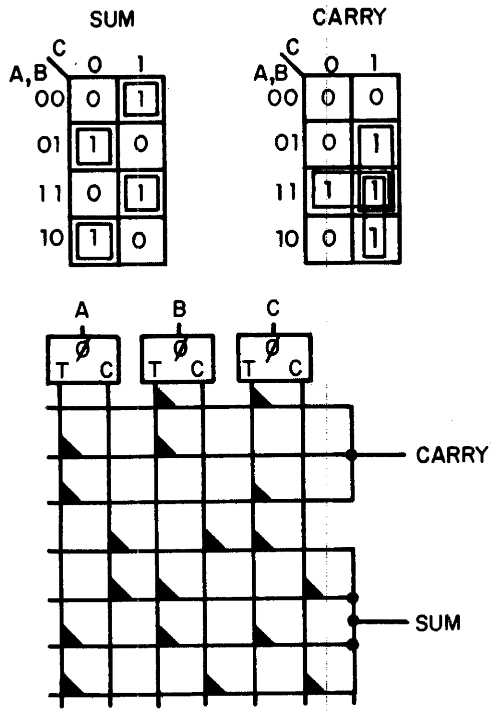


Figure 6—ROAM adder position

are DOT-ORed at the output to yield B · C + A · B + A · C = CARRY. In forming the term A · B, the variable C does not have its true or complement column line connected to a base. CARRY is 1 if A is 1 and B is 1 regardless of the value of C.

Each term of a logical expression in sum-of-products form is an "implicant" on a Karnaugh Map. An implicant is formed by looping the 1's in the Karnaugh map and "reading" the loops from the map. Loops can only contain adjacent 1's and the number of ones in a loop must be equal to 1, 2, 4,..., a power of 2. This results from the fact that adjacent squares on a Karnaugh map always differ only by the value of one variable. Two squares looped yields a term with n-1 variables (n = number of variables), four squares looped yields a term with n-2 variables, etc. Thus, each implicant requires one row in a ROAM. The bigger the loop of 1's the fewer connections need be made in that row. The complete expression is formed by DOT-ORing the rows which is the same as ORing the implicants.

The example of Figure 6 uses three loops of two 1's each to form the CARRY. The SUM is formed by four loops of one 1 each. In this case three con-

TABLE I—Bits required for n variables in ROS and ROAM ARRAYS

| | VARIABLES | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 2 | 3 | 4 | 5 | 6 | 7 | 8 | n |
| | BITS | | | | | | | |
| ROS Always Universal | 4 | 8 | 16 | 32 | 64 | 128 | 256 | $2^n$ |
| ROAM | | | | | | | | |
| *Implicants (Rows)* 2 | 8 | 12 | 16 | 20 | 24 | 28 | 32 | 4·n |
| 3 | | 18 | 24 | 30 | 36 | 42 | 48 | 6·n |
| 4 | | 24 | 32 | 40 | 48 | 56 | 64 | 8·n |
| 5 | | | 40 | 50 | 60 | 70 | 80 | 10·n |
| 6 | | | 48 | 60 | 72 | 84 | 96 | 12·n |
| 7 | | | 56 | 70 | 84 | 98 | 112 | 14·n |
| 8 | | | 64 | 80 | 96 | 112 | 128 | 16·n |
| 9 | | | | 90 | 108 | 126 | 144 | 18·n |
| 16 | | | | 160 | 192 | 224 | 256 | 32·n |
| $2^n/2$ Rows (Universal) | 8 | 24 | 64 | 160 | 384 | 896 | 2048 | $n \cdot 2^n$ |

nections must be made in each of the four required rows to obtain

$$\mathrm{SUM} = \overline{\mathrm{A}} \cdot \overline{\mathrm{B}} \cdot \mathrm{C} + \overline{\mathrm{A}} \cdot \mathrm{B} \cdot \overline{\mathrm{C}} + \mathrm{A} \cdot \mathrm{B} \cdot \mathrm{C} + \mathrm{A} \cdot \overline{\mathrm{B}} \cdot \overline{\mathrm{C}}$$

In contrast to the ROS, the ROAM can have universal capability with only one-half the number of rows as the ROS needs bits for the same number of variables. Moreover, the ROAM does not need to be universal to be useful, thus allowing even further reduction in size. Table I illustrates the difference brought about by the ROS requiring one bit per K-map position and the ROAM requiring one row per K-map implicant.

Historically, computer functions are composed of about four implicants or terms. The chart shows that a four-implicant function is cheaper to implement with a ROAM than with a ROS when the function contains six variables or more. When the decoders required for the ROS are considered, even four-variable functions with four implicants are more economical in ROAM than in ROS.

Two useful formulas to compare ROS bits required with ROAM bits required for a given function are:

$$\text{ROS bits} = 2^n$$

$$\text{ROAM bits} = 2\,I\,n\,,$$

where $n$ = number of variables, $I$ = number of implicants. Thus, it is more economical to build a function with the ROAM when $2\,I\,n < 2^n$. This does not consider the cost of the ROS decoders, which add a factor to the inequality.

If we assume that the decoders for n-even take $2n(2^{n/2})$ bits, and for n-odd take $[(n+1)\,2^{(n+1)/2} + (n-1)\,2^{(n-1)/2}]$ bits, then the cases for which ROAM should be used are:

1. $n$ even
   $2\,I\,n < 2^n + 2\,\mathrm{n}(2^{n/2})$;
2. $n$ odd

   $2\,In < 2^n + (n+1)\,(2^{[n+1]/2}) + (n-1)\,(2^{[n-1]/2})$

Thus, ROAM is more economical than ROS in most practical problems.

A realistic example of control logic for a small machine model has been implemented using the ROAM array. Table II gives a comparison of the number of bits required for a ROAM implementation versus the number of bits required for a ROS implementation. Note that the ROAM is significantly more economical.

A partitioning of functions could have been devised for the ROS implementation. The ROAM would still

TABLE II—ROS vs. ROAM—a control logic example

| | | |
|---|---|---|
| TOTAL NUMBER OF VARIABLES | | 14 |
| TOTAL NUMBER OF FUNCTIONS | | 6 |
| TOTAL NUMBER OF IMPLICANTS | | 12 |
| | One 7-implicant function of 13 variables | |
| | Four 1-implicant functions of 7 variables | |
| | One 1-implicant function of 11 variables | |
| *ROAM* | | |
| | ARRAY SIZE: $28 \times 12$ | 336 BITS |
| *ROS 1* | | |
| | ARRAY SIZE/FUNCTION: $2^{14}$ | 16,384 BITS |
| | 6 ARRAYS FOR 6 FUNCTIONS: $6 \times 16{,}384$ | 98,304 BITS |
| | SHARED DECODER | 3,584 BITS |
| | TOTAL BITS | 101,888 |
| *ROS 2* | | |
| | ARRAY SIZE FOR 13 VARIABLES: $2^{13}$ | 8,192 BITS |
| | ARRAY SIZE FOR 7 VARIABLES: $2^7 \times 4$ | 512 BITS |
| | ARRAY SIZE FOR 11 VARIABLES: $2^{11}$ | 2,048 BITS |
| | SHARED DECODER | 3,584 BITS |
| | TOTAL BITS | 14,336 |

be more economical than the ROS, however, especially when one considers the additional wiring complication of connecting several small ROS arrays and the additional design time required to effectively partition the functions.

The optimum size for a ROAM has not been determined, but chips with at least 512 bits on them are desirable. This capacity would provide between eight 8-variable, 4-implicant functions, and one 64-variable, 4-implicant function (an extreme case, needless to say) on a chip. The practicality of building and using such a chip is yet to be determined.

## The SLT array

Arrays can be designed so that they may be used for direct replacement of present logic. The SLT array performs the function AND-OR-INVERT in negative logic or OR-AND-INVERT in positive logic and can be used directly to replace SLT logic. While direct replacement of random logic with array chips may prove to be the wrong approach in the long run, it may well be the only way to get array logic started.

The SLT array has the same advantages over ordinary logic that all arrays have: orderliness of design and layout, and high density with relatively low cost.

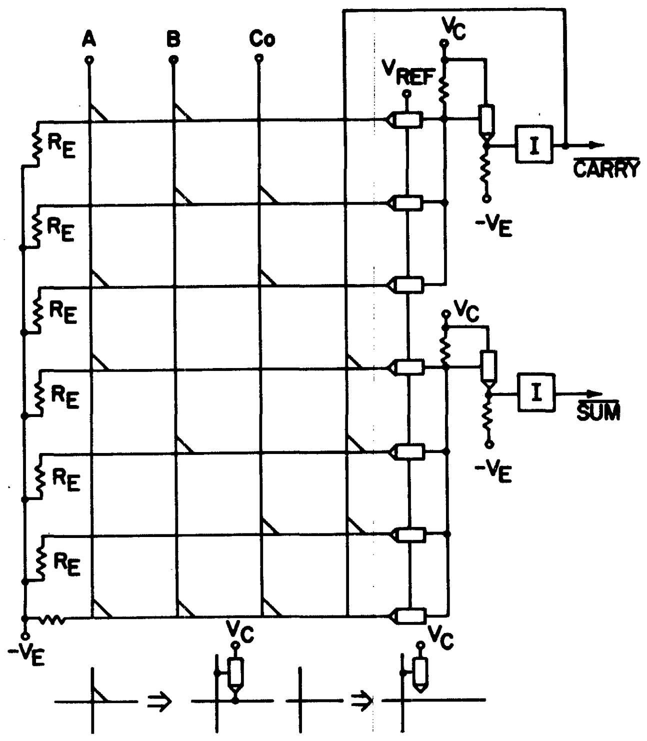


Figure 7—SLT array

In addition, this type of array has a higher bit usage than other arrays, since it more closely resembles the familiar random logic, functionally. The SLT array does not have decoders or phase splitters on its input lines, as do other types of arrays. This makes the array less universal than even the ROAM array but more effective for random logic. It is fair to say that arrays of this type make poor code translators just as SLT logic builds poor translators. It is difficult to believe that any array will be effective in both random logic and code translation problems.

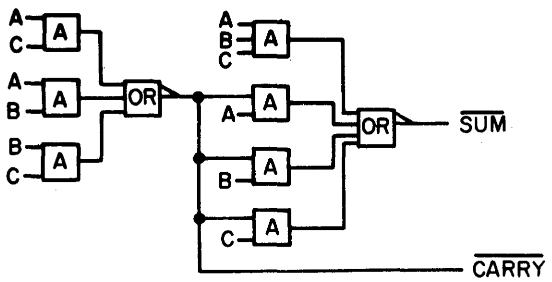
As already stated, the ROAM array has specific applications to decoders and associative memory problems. The SLT array may very well be the element required to do general logic design. The reason for this is the placement of the inverters as shown in Figure 7. This movement of the inverters to the output lines may appear a minor modification, but it should be remembered that there has never been a useful logic block with inverters on the input lines. It may pay to have both true and complemented outputs from a current switch logic block. Figure 8 shows a full adder implementation in SLT logic and in an SLT array.

## Array-driving arrays

The SLT array in Figure 8 demonstrates one necessary feature of an array that has yet to be discussed: Any logic array must be able to drive any other array in the same family, including itself. Note in Figure 8 the $\overline{\text{CARRY}}$ output fed back into the array. This line probably will be an external wire. This technique is required since it is in effect Boolean factoring, a proven necessity. This type of feedback is also needed to produce sequential circuits, giving memory to the arrays.

## Figure of merit

It is less meaningful to compare array logic with random logic in each individual term of power consumption, propagation delay time, and silicon area, since one can usually be traded for the other, such as power with delay. Instead a comparison is made of their figures of merit, chosen to be the product of power consumption P, delay time T, and silicon area A, all with weight function of one (PTA). Since no isolation wall is needed between collector transistors, a ROS or ROAM cell including appropriate interconnections can be laid out on a silicon chip area equivalent to 20-25 percent of that occupied by a transistor that needs isolation walls. As shown in Figures 5 and 7,



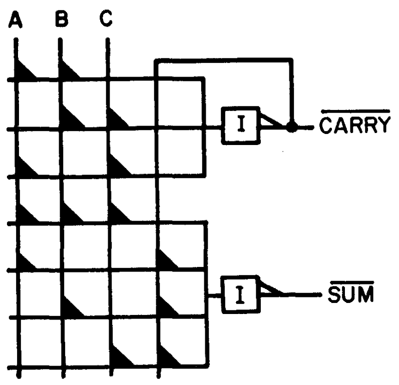


Figure 8—SLT full adder position

the delay time of an array is two levels of current switch emitter follower (CSEF) independent of the number of inputs. For sophisticated functions, such as the one-bit adder shown in Figure 8, more than two levels of logic may be required.

Some typical comparisons of array logic and random logic include the sampling design of array logic chips to perform the same function a random logic chip would. This comparison helps to partially discover the merit and the limitation of the array logic. In comparison with random logic chips that perform sophisticated functions or have two or more cascading levels of CSEF's, array logic chips have superior PTA figures.

## CONCLUSIONS

Various array configurations described here suggest that random logic may be implemented by use of an array of programmable crosspoints. Comparisons of array logic with conventional logic indicate that in many cases the PTA figure of merit is superior for arrays. The most significant problem with arrays appears to be the limited useful size of a single array, and the difficulty in standardizing a particular array configuration. As a minimum achievement at this time, it appears that arrays will be useful in development of complex functions within a silicon chip.

Array logic will not eliminate the need for a circuit designer in the future, since specialized designs will be needed to optimize circuit and component technology. In some of these design cases, the importance of array logic techniques will be obvious, but in others it will not be.

At this point, array logic does not appear to strongly affect the system designer's approach to machine design, and a knowledge of array logic may never be required.

In the future, however, to the extent that array logic techniques influence the design and optimization of highly efficient functions, the system designer's work will be significantly influenced by progress made in developing array logic techniques.

## BIBLIOGRAPHY

1 R RICE
*Computers of the future*
IBM Research Report RC-151 April 20 1959
2 R RICE
*Systematic procedures for digital system realization from logic design to production*
Proc IEEE Vol 52 12 1691-1702 Dec 1964
3 R C MINNICK
*Application of cellular logic to the design of monolithic digita systems*
Microelectronics and Large Systems
Spartan Books Wash D C 1965 225-247
4 L C HOBBS
*Effects of large arrays on machine organization and hardware software tradeoffs*
Proc FJCC 1966 Vol 29 89-96
5 R C MINNICK
*Cutpoint cellular logic*
IEEE Transactions on Electronic Computers Dec 1964
6 W E KING III A GUISTI
*Can logic arrays be kept flexible?*
AFCRL Report 65-547 Aug 1965
7 D C FORSLUND R WAXMAN
*The universal logic block (ULB) and its application to logic design*
IEEE Conference Record 1966 Seventh Annual Symposium on Switching and Automata Theory 236-250
8 S S YAU C K TANG
*Universal logic circuits and their modular realization*
Proc SJCC 1968
9 R C MINNICK
*A survey of microcellular research*
Jour ACM Vol 14 2 April 1967 203-241

# Characters—Universal architecture for LSI

*by* F. D. ERWIN and J. F. McKEVITT

*Hughes Aircraft Company*
Fullerton, California

## BACKGROUND

Since the advent of LSI technology, several schemes have evolved for the utilization of large arrays to their full potential. A common and straightforward approach involves the designer restricting himself to the equipment being designed at the moment. Faced with only a limited set of problems, it is not difficult to specify a small number of LSI array types which will efficiently complete the design. While the results are quite encouraging for specific cases,[1] the drawbacks of any mass adoption of these techniques are obvious. This, the so-called "custom approach," would require the semiconductor manufacturer to be responsive to each customer with numerous low-output production runs of highly specialized devices. The per-unit cost to the user, for his own efforts as well as those of the manufacturer, would be quite high due to the inability to spread initial costs over many devices. In addition, the complexity of 100-gate-plus arrays is such that it is difficult to substitute one for another (with efficient results). This would severely limit the off-the-shelf capabilities of both user and manufacturer.

An obvious solution to these problems is the introduction of a small set of standard LSI chips. Semiconductor suppliers, making tentative advances into LSI product marketing, have already proposed such devices as adders, counters, and shift registers. However, this does not represent the solution to the general problem. A design heavily committed to the use of these devices must fall back on MSI or standard IC for the large remainder of the circuitry. The reason is that adders, counters, registers and other orderly, well-defined areas represent the regions of the system with the highest gate-to-pin ratios. After these portions are lifted out of the system, the remainder is characterized by very low gate-to-pin ratios (notably control and data routing functions). Unable to satisfy the LSI design criteria of high gate-to-pin ratios any longer, the designer must look to more standard components. Unfortunately, any proposed solution to the LSI partitioning problem which lacks a total system approach tends to drift towards this pitfall.

Researchers striving towards partitioning for total or near-total LSI implementation tend to diverge along one of two conceptual paths; bit-slicing and functional partitioning. To illustrate the difference, consider the data portion of the computer. In functional partitioning one may specify an adder as one LSI array, registers as another, a shift register as a third, and so forth. On the other hand, in bit-slicing one would design an LSI array consisting of a combined one- or two-bit adder, registers, shift registers, etc., then build up his system from this chip type according to the desired word length.

The bit-slice approach has resulted in some notable advantages, particularly the ability to achieve very high gate-to-pin ratios and implement systems using a small number of different array types.[1,2] However, bit-sliced modules have the basic flaw of being system-dependent, a drawback described by Pariser in an early paper.[3] This means that behind such bit-slicing approaches there lie systems, real or implied, for which the resulting arrays are most efficient. An attempt to apply the arrays to a significantly different system results in a poor design. Considering the types of bit-slice devices being proposed, inefficiencies would most often be manifest in the design of a simple device in which the majority of the gates of the array intended to accomplish complex functions are wasted. Although this may be acceptable in some situations, it is unlikely that it would satisfy the strict requirements of size, weight, power, and reliability imposed by aerospace and military systems.

It is the contention of this paper that a judicious partitioning of digital systems in general, divorced from bias towards any particular system, results in a set of LSI devices that can entirely implement many different computer systems of varying functional complexities and word lengths.

The resulting group of arrays, referred to as a "character set" and each one individually as a different "character", is sufficiently small in number (10), with each type having acceptable size and gate/pin ratio, to be considered acceptable and desirable in view of its wide range of applications. These building blocks are referred to as characters because of the metaphor that may be made between the building blocks and characters of the alphabet (letters). Letters form words to express the language whereas building blocks form units to build the machine. In both cases a closed set (of characters) is used to produce the desired end.

Although the character set is neither rigidly functionally-partitioned nor bit-sliced, it is biased towards functional partitioning to give it the versatility to efficiently implement both complex and simple digital devices. As an approach, functional partitioning has a detailed and successful background.[3,4] Bit-slicing consideratoins give the character set its ability to implement systems of varying word lengths.

In addition to providing the user with a standard set of chips to implement many different digital machines, the completeness of the approach (the ability of the characters to implement the whole machine) relieves the user of the burden of logic design. These tasks are reduced to the selection of character types and word lengths.

## *Introduction to the character set*

A universal conclusion among LSI researchers is that control functions are more difficult to modularize than functions related to data operations. Micromemory control technique was chosen as the solution for LSI implementation for several reasons. A micromemory, meaning here a read-only solid-state memory with its sequencer and instruction register, is easily partitioned into the large modules necessary for LSI implementation. Control functions in this form are then amenable to reproduction in large quantities of identical units. Also, design with control centered in one level of micromemory is more orderly and straightforward.

The micromemory has been provided with a relatively sophisticated microprogram instruction repertoire. This means that the microprogram contains the essence of the machine's major mathematical functions, such as multiply and complex sequencing. This is desirable since it represents an efficient use of hardware for these purposes and also reduces the number of different array types necessary. Also, a versatile repertoire leaves the designer free to make units which operate as simply or as complexly as desired. The degree of flexibility which this repertoire gives the character set is a major factor in its success. It should be stressed that the "micro operations" of the character set are as important a factor as its logic design. This fact, a critical one in all LSI solutions committed to micromemory control, cannot be overemphasized.

Interest in designing a character set at Hughes was concurrent with the development of an advanced computer system. The character set itself was developed with the ultimate objective of implementing all future Hughes digital data processing equipment with a common family of LSI circuits.

The outcome of that original effort revealed that computer structures in general are frequently ordered, or at least amenable to such ordering, as shown in Figure 1.

The divisions of Figure 1 are functional. That is, regardless of the hardware characteristics, the computer philosophy is such that its functions may be identified, separated, and diagrammed as shown in the figure.

From Figure 1 came the concept of the functional character set. With the fundamentals of LSI design in mind, logic was designed to accomplish each computer

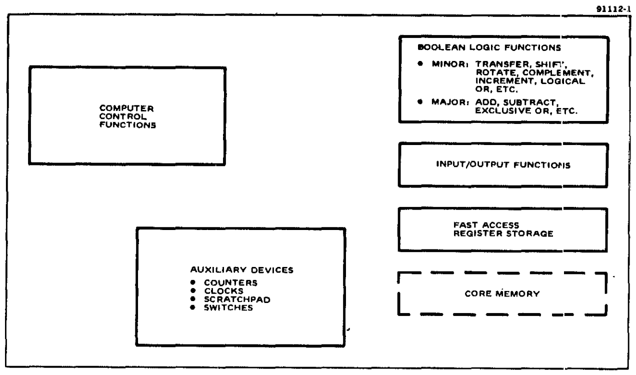

91112-1
BOOLEAN LOGIC FUNCTIONS
• MINOR: TRANSFER, SHIFT, ROTATE, COMPLEMENT, INCREMENT, LOGICAL OR, ETC.
• MAJOR: ADD, SUBTRACT, EXCLUSIVE OR, ETC.
COMPUTER CONTROL FUNCTIONS
INPUT/OUTPUT FUNCTIONS
FAST ACCESS REGISTER STORAGE
AUXILIARY DEVICES
• COUNTERS
• CLOCKS
• SCRATCHPAD
• SWITCHES
CORE MEMORY

Figure 1—Computer functional organization

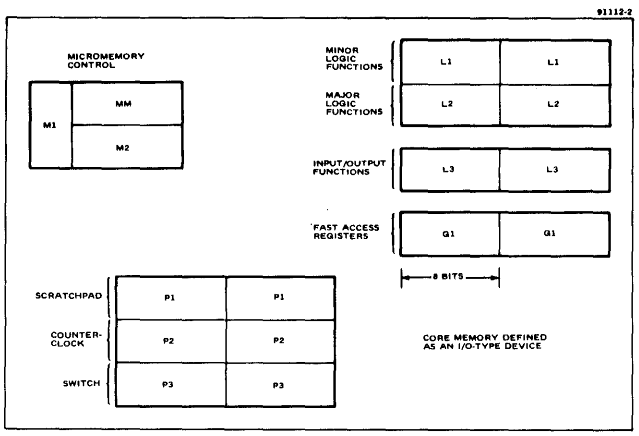


Figure 2—Functional character set

function indicated by the picture. Each unique LSI chip type which resulted was referred to as a different character type and given an identifying name and number. Figure 2 shows the character set which resulted from the logic design according to the concepts outlined in Figure 1.

The character set and repertoire have been through several improvement cycles and used in the test implementation of a NASA computer to be discussed later. Current plans include test design of the H4400 (a new Hughes computer) with the improved character set, implementation of the character set with high speed MOS circuits, and construction of one computer using the characters.

These ten LSI characters alone provide the entire hardware complement for the logic of a broad range of computers and digital equipment. No extra logic in the form of either IC, MSI, or custom LSI need be added to the characters to finish the job. An important by-product of this is that the user need never consider logic design. His tasks are reduced to selection of the necessary characters and the writing of the appropriate microprograms for them. In fact, it is possible for the character set to fit into a realistic total design automation procedure as discussed later.

### *Description of the character set*

This section describes each of the ten characters. They are summarized below for reference.

| | |
|---|---|
| G1 | Register storage |
| L1 | General logic |
| L2 | Arithmetic logic |
| L3 | Input/Output |
| M1 | Micromemory counter |
| M2 | Micro-instruction Register |
| MM | Micro-array |
| P1 | Scratch pad memory |
| P2 | Up/Down counter |
| P3 | Switch |

Characters of the same letter are logically grouped into a common unit as illustrated in Figure 3.

### G1 character

The G1 character provides the bulk of storage for operands of the microprogram. Each character contains four registers of eight bits each accompanied by reading and writing selector gates. The storage element is provided with simultaneous dual reading and writing capability. The storage flip flop itself is designed for minimum read after write delay.

Each of the two input busses is common to all registers and carries to the G1 character eight lines per bus, one line from each bus for each bit of the register. Input data selection is accomplished at the memory element by a coincidence of positive information on a particular input bus and register selection for that bus by destination decoding logic within the character. The destination decoding logic is duplicated to provide for writing from the two input busses into the same character under control of two different microcommands. As will be illustrated later, this is a key factor for the machine expandability property of the character set as it allows G1 to form a data path link between individual logic units under control of up to two different micromemories. Different registers in the character may be written into simultaneously.

Reading of the register is provided by dual source decoding logic which gates data to independent dual output busses. This duality provides for information from any two registers to be simultaneously placed on two output busses. The conceptual structure of the G1 character is shown in Figure 4.

Several G1 characters placed in parallel provide registers of more than eight bits in length.

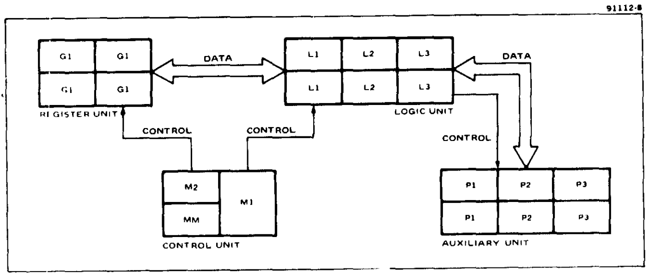


Figure 3—Typical functional character configuration

## L1 character

The L1 character provides the basic logic functions selectable by microprogram. In addition input bussing is provided for nine channels (eight bits/channel). One channel of the bus is required for each G1, L2 or or L3 character connected to the L1 character. The logic functions provided consist of the rotates, shifts (logical), no-operation, complement, and incrementation. Also associated with the L1 character is the decoding logic for these logic operations. The type of microprogramming used with the functional character system relies heavily upon the fast and efficient manipulation of bits within the various operands. To this end, shifts and rotates have been provided which execute from 1 to 31 positions in a single step (as opposed to serial operation). Incrementation is accomplished with the use of a logic register which may also be used as a simple holding register. The L1 character is eight bits wide and contains the following logic:

1. Bussing gates
2. Decoding logic
3. Rotate, shift, and complement logic
4. Incrementer
5. L register
6. Gating to output bus

In Figure 5 is shown a block diagram of the L1 character. Several L1 characters may be connected together to form logic operations on words longer than

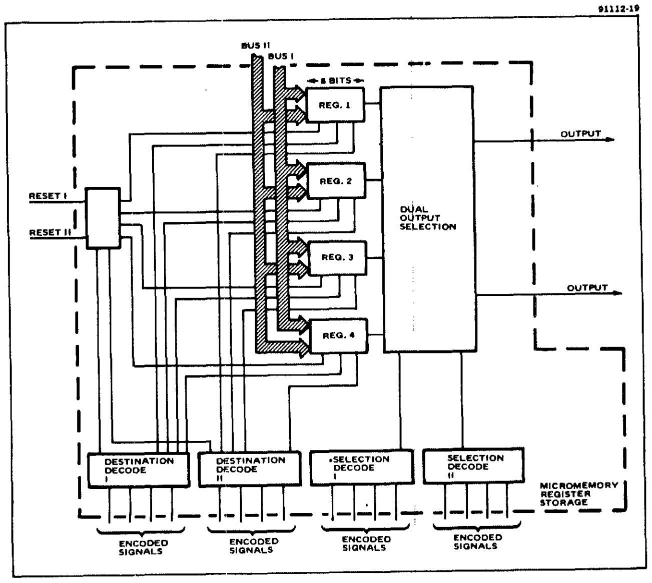


Figure 4—G1 character block diagram

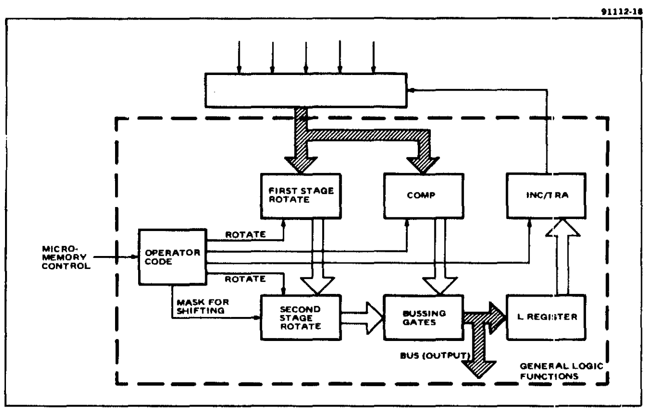


Figure 5—L1 character block diagram

one byte. A limit of four bytes exists in order to maintain consistency of definition in the rotates and shifts.

Information entering the L1 card from the various sources is bussed to form the input bus. Then it is operated upon and the resultant is bussed to the output bus where it leaves the character or is optionally stored in the L register (where it would thus be available at the next micro-instruction time for use in the increment operation or as an "L" source).

## L2 character

The L2 character provides the major arithmetic functions used by the microprogram. The arithmetic unit provides the 2's complement sum of the contents of the A and B registers. Addition is performed with carry look-ahead byte parallel. Control signals may condition the adder to alternately provide either of two special results (a) a mod 2 addition instead of full addition or (b) an input carry to the lowest order bit for full addition (this forced carry in conjunction with a negated operand accomplishes a 2's complement operand for subtraction). The L2 character consists of two holding registers for the operands of the adder, the adder itself, decoding and error logic, and bussing gates. Figure 6 diagrams function-wise the L2 character.

A typical arithmetic operation using the L2 character might proceed as follows: (1) first operand transferred to B register (from output bus), (2) second operand transferred to A register, (3) after appropriate delay access result and transfer out of L2 character via the input bus. The error logic provides overflow and carry-out information.

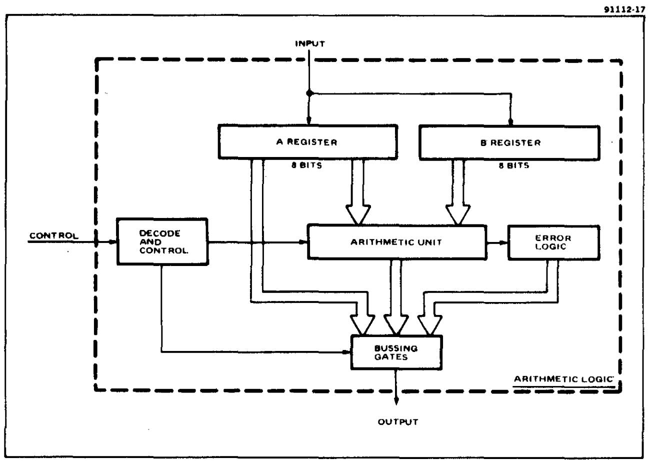


Figure 6—L2 character block diagram

## L3 character

The L3 character provides input/output capability for the microprogram machine. For purposes here input/output includes not only the usual peripherals but also main memory, scratch pads, real time clocks, all P-characters—namely all elements of the computer not directly controlled by the micromemory. The L3 character provides input gating for external devices—four buffered and three non-buffered channels. The buffered-input gating may be controlled either by the microprogram or the external I/O device itself. Four I/O output channels are provided. Interrupt signal storage and interrupt mask storage for four channels are available. Parity generation and checking along with odd/even control is provided for the four buffered channels. L3 also contains the necessary register destination and selection logic. Figure 7 is a block diagram of L3.

To input data, an input line is selected under microprogram control resulting in selected data entering an E register or, in the case of a non-buffered input, entering the input bus. To output data, the micromemory places the data in the appropriate E register and signals the corresponding I/O unit. The E registers themselves are available to the logic unit in a manner identical to the G registers (G1) independent of their input/output functions.

## M1 character

The M1 character provides the micromemory address register and related functions. The ten address bits of M1 allow for addressing up to 1024 micromemory words. The address is contained in the MMC (Micro

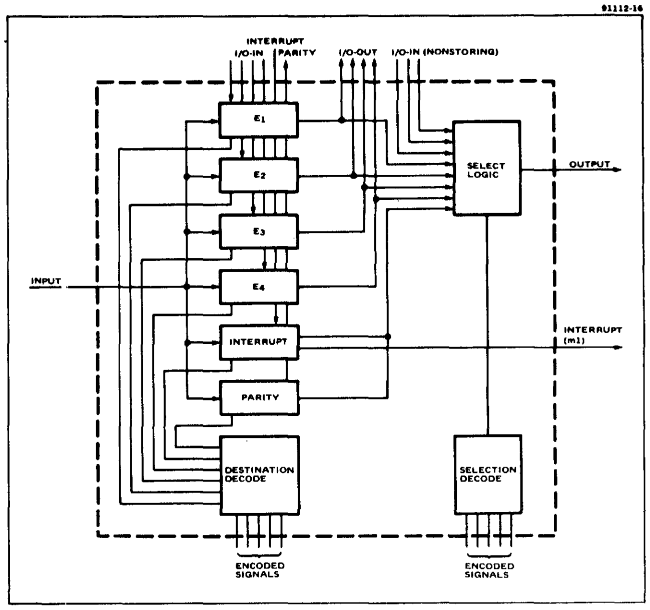


Figure 7—L3 character block diagram

Memory Counter) register and serves to address the micromemory proper. Associated with the MMC register is a five-bit incrementer which automatically steps through 32 microprogram address states and then repeats addresses. This produces the effect of a microprogram ring of 32 words in which the program will loop until the microprogram issues an unconditional transfer command. There is an S (save) register that allows for subroutine jumps. The S register saves the content of MMC upon command, keeping it available for reinsertion into MMC. Figure 8 shows the block diagram for M1.

Branching or transferring within the microprogram is provided by two modes: unconditional transfer of full 10-bit width and conditional transfers of four bit

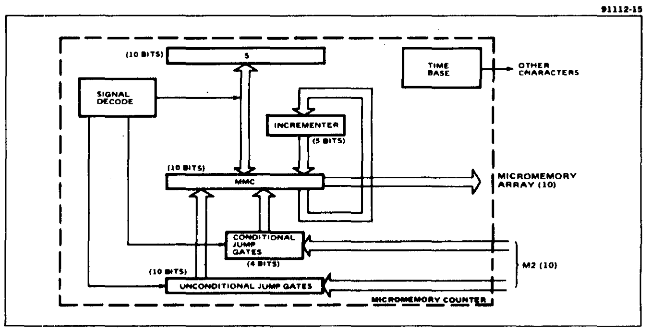


Figure 8—M1 character block diagram

width. The M1 character carries the time base whose signal is distributed to other characters.

### M2 character

The M2 character contains a micromemory word register. The register is 49 bits long providing for a full micromemory word. Forty-nine bits are divided into two 16 bit fields and a 17 bit field. The first and the second fields are instructions and the third is a constant. The second instruction is transferred into the register location of the first for execution resulting in sequential execution of the two instructions in the micromemory word. Timing is derived from the timing base on the M1 card. Figure 9 shows the block diagram of M2.

### MM character

The MM character contains the micromemory array. The address register and word register for the array are located on M1 and M2 respectively. MM is a read-only array. The presence of an address on the input lines causes the contents of the referenced location to appear on the output lines after an appropriate delay. The MM character consists of 256 words of 49 bits each. Figure 10 shows the block diagram of MM.

Several MM characters may be combined to form a larger micromemory array. The maximum organization is 1024 words by 98 bits.

### P1 character

The P1 character is a scratch pad memory of 256 bits of storage with associated address decode logic, address register and data register. The scratch pad is arranged into 16 registers of 16 bits each. Figure 11 is a block diagram of P1.

The P1 character is connected to the L3 character through which its data flows. Up to 16 P1's may be connected in series to produce a total scratch pad of 256 registers. Generally the bit width will match that of the logic unit.

### P2 character

The P2 character is an expandable eight-bit counter with byte look-ahead logic. The introduction of a time signal produces a real-time binary clock. The counter may be read in parallel and is resettable to any desired value. Zero detection is provided which may optionally interrupt the microprogram and/or the main program. The P2 character is connected to the L3 character through which data and control pass. Figure 12 shows the block diagram detail.

The P2 character contains control logic allowing the counter to be in a run state or stop state dependent upon microprogram control.

### P3 character

The P3 character provides the capability of switching any three input channels to any three output channels.

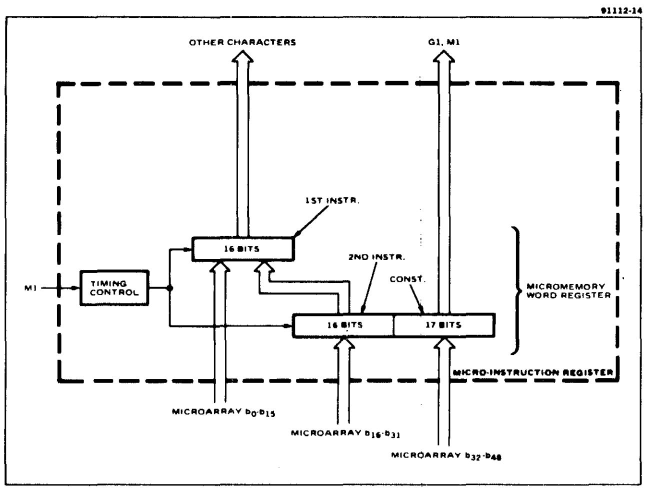

91112-14
OTHER CHARACTERS
G1, M1
1ST INSTR.
16 BITS
2ND INSTR.
CONST.
M1
TIMING CONTROL
MICROMEMORY WORD REGISTER
16 BITS
17 BITS
MICRO-INSTRUCTION REGISTER
MICROARRAY $b_0$-$b_{15}$
MICROARRAY $b_{16}$-$b_{31}$
MICROARRAY $b_{32}$-$b_{48}$

Figure 9—M2 character block diagram

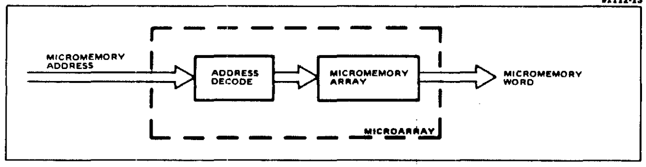


Figure 10—MM character block diagram

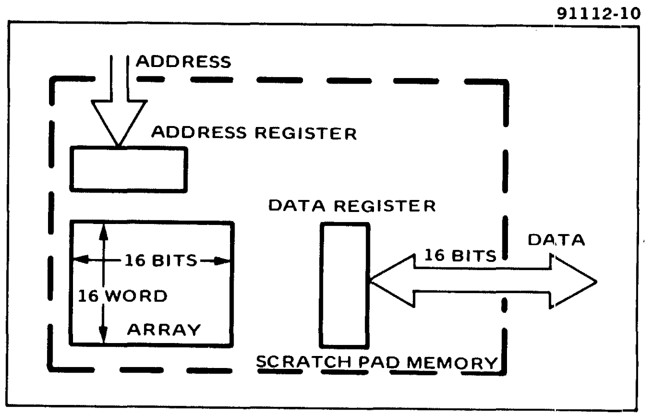


Figure 11—P1 character block diagram

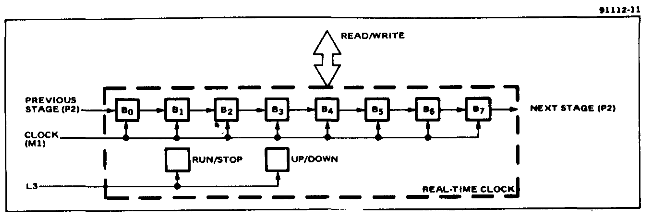


Figure 12—P2 character block diagram

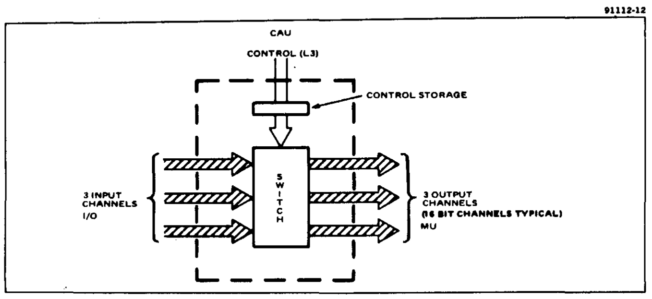


Figure 13—P3 character block diagram

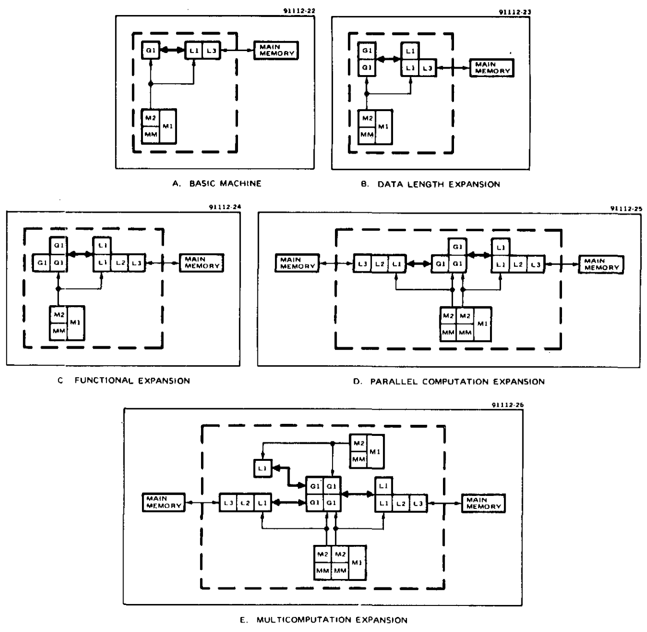


Figure 14—Four stages of expandability

A 16-bit width is provided. This configuration allows three simplex simultaneous connections. Figure 13 shows the block diagram for the switch.

The input and output channels of P3 may be connected to any external interfaces which are electrically compatible. Storage is provided on the character for nine bits of control information establishing the state of the switch.

There is no restriction on the switch state; all possible configurations are allowed (such as three inputs to three outputs, one input to three outputs, three inputs to one output, etc).

### *Hardware applications*

Provided these ten characters and given a design performance specification, the decisions the designer must make involve considerations of character types and selection of word lengths.

Figure 14 illustrates the levels of machine complexities available to the designer. Part A illustrates a very basic eight-bit machine, with simple logical, I/O, and register capabilities. Part B is the machine of part A expanded to 16 bits in its logic and register portions; however, no new functional capabilities have been added. Functional expansion is demonstrated in part C, where an eight-bit adder card and four eight-bit registers are added. Part D represents a significantly greater jump. Illustrated is the dual-logic unit capability of the character set. If desired, it is possible to have two logic units, with different but coordinated microprograms, operating in parallel. They share the same sequencer (M1), which both control. The G1 bank is common to both logic units.

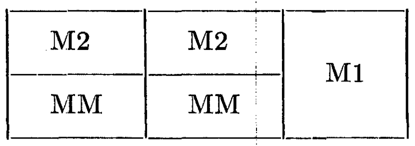
Part E illustrates an even higher level of expansion. Two totally independent micromemory units (memory and sequencer) drive three different logic units, linked together through G1 cards. This level of complexity can be carried to an almost limitless expansion of micromemories and logic units bound together by shared G1 characters. A comparison of parts A and E of Figure 14 illustrates the versatility of the character set as it is adapted to both simple and complex situations.

With the hardware specified, the next major task is the writing of microprograms. As stated before, in machines of this type this is as important as the hardware design. Often the only essential difference between units designed for different purposes is their microprograms.

The microprogram repertoire designed for the character set is described in the next section.

### *Description of microprogram repertoire*

The micromemory word provides the control necessary for the functions of the characters under its direct influence. All these characters so controlled are defined to belong to a common instruction group. There is one and only one M2 character per instruction group. A phase group consists of usually one or two co-instruction groups containing a common timing base. There is one and only one M1 character per phase group as illustrated below.



In a phase group containing two instruction groups one micromemory word, accessed from the first micromemory array (MM), operates upon and through its logic unit while the other word, accessed from the second micromemory array (MM), operates upon a second logic unit. Operations are carried out simultaneously in each unit with some cross translation. The option of including a second micromemory word allows for greater system capability by providing simultaneous operations; however, this does not affect the number of bits in the data word. (The data width is independently variable by byte.)

A micromemory word is composed of two 16-bit fields and a 17 bit field—two instruction fields and a constant field (See Figure 15). The first and second instruction fields are identical differing only in that execution of the second instruction follows the first by 1/2 of cycle time (a cycle time is the time required for a complete cycle of the micromemory). The instructions can access the constant field, introducing into the data stream this constant from the micromemory. At those times when the constant field is not used as such, it takes on additional capability as a transfer and machine control field.

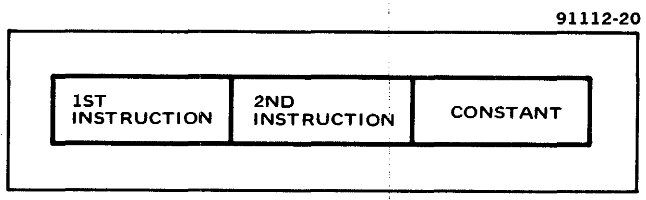


Figure 15—Micromemory word

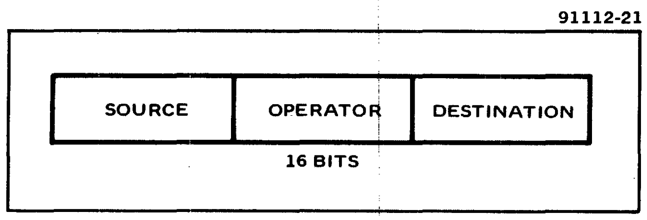


Figure 16—Instruction field

Instruction Fields—Each instruction field is divided into three subfields—source, operator, and destination subfields as shown in Figure 16.

The source specifies the origin of the data to be operated upon as defined by the operator field. The destination specifies the location where the data result will be stored after the operation is performed.

Source Subfield—The source subfield specifies the source of information for the micro-instruction. Data accessed by the source code appears on the input data bus. Typical sources are:

| | |
|---|---|
| G1 -G16 | —The general set of registers |
| E1 -E12 | —The I/O registers located on the L3 characters |
| CNT | —The constant field from the micromemory word |
| INC | —The incremented value of the L-register |
| A | —A register located on card L2 |
| ADD | —The sum from the L2 character |
| L | —The L register of the L1 character |
| ECS | —The error code |
| *ADD | —The sum from the L2 character of a co-instruction group logic unit |

Operator Subfield—The operator subfield specifies the type of operation the micro-instruction involves. These operators operate upon the data from the input bus and present the result at the output bus. Typical operators are:

| | |
|---|---|
| RS1 — 31 | —A Right Shi'ʼt from 1 to 31 positions |
| LS1 — 31 | —A Left Shift from 1 to 31 positions |
| MSK | —The source data masked by the constant field of the micromemory word |
| NØP | —The no-operation |
| R1 -R31 | —A left rotate from 1 to 31 positions |
| CØM | —The ones complement |

Destination Subfield—The Destination Subfield specifies directly the register to receive the instruction result. These register designations are described below:

| | |
|---|---|
| G1 -G16 | —The general set of registers |
| E1 -E12 | —The twelve I/O registers of the L3 characters |
| B | —The B register of the L2 card |
| L | —The logic register of the L1 card |
| A | —The A register of the L2 card |
| *A | —The A register of a co-instruction group L2 character |

Transfer Field—The transfer field allows for microprogram specification of both conditional and unconditional transfers within the microprogram. The unconditional transfer provides a ten-bit address, the full microprogram addressing capability, while conditional transfers provide four-bit addresses. At all times when a transfer is not effected (either conditional or unconditional) the micromemory counter is incremented by one modulo 32.

There are basically three testable functions. They are: (1) least significant bit—true; (2) most significant bit—true, and (3) all bits—false (true = 1, false = 0). Further, some of these functions may be tested as inputs to the logic unit or as outputs and in various combinations.

There exist eleven conditional transfer test combinations and one unconditional transfer.

### *An application of the character set*

In addition to investigation for use with the H4400, the Hughes Character Set was used in a test design of the NASA Modular Computer Breadboard (MCB). The NASA MCB, a prototype of an advanced aerospace computer, is a dual-redundant reconfigurable machine consisting of five different module types. One each of the Control Unit (CU), Arithmetic Unit (AU), Memory Unit (MU), and Input-Output Unit (I/O) are required for a working computer. The fifth module type, the Configuration Assignment Unit (CAU), is not duplicated. For a detailed description of the NASA MCB, see "Implementation of the NASA Modular Computer with LSI Functional Characters," by Pariser and Maurer, in these Proceedings.

Figure 17 shows how the NASA MCB can be implemented using the Hughes character set. Notice that the CU is the only module equipped with the double-logic unit feature.

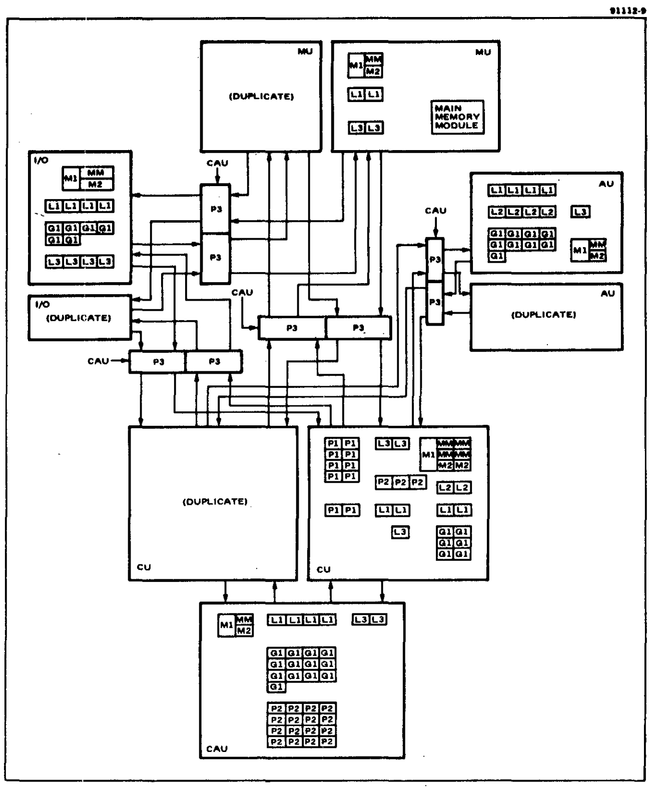


Figure 17—MCB-Modular computer breadboard block diagram

The design of the NASA MCB showed that a fairly complex computer could be implemented using only the ten characters. Comparison of the gate counts with that of a computer built to similar specifications indicates that design with the character set involves approximately 35 percent more gates (exclusive of ROM). The comparison machine was composed of 23 different card types contrasted to the character set's ten. The overall gate-to-pin ratio was 2.6 for the character set version and 0.75 for the comparison machine.

Table I is a representative sampling of the estimated MCB instruction execution rates. By these estimations, the character set version is capable of running as much as 55 percent faster than the prototype machine. A large part of the speed and versatility of the MCB were attributed to the total microprogram approach of the character set. Since each unit has its own micromemory control, it was possible to utilize unit overlap to the maximum advantage.

Performance specifications for machines built from the character set assume the following about the characters themselves. Each character involves approximately 300 gates based on SUHL II type logic. Most characters may be sub-partitioned into two identical LSI wafers of 150 gates each. Gate-to-pin ratios[5] vary from character to character with an overall average of about 2.6. Each level of gating must involve a propagation delay of no more than 12 nanoseconds to achieve the indicated speeds. Read-only-memory access time is assumed to be no more than 80 nanoseconds with a cycle time of 200 ns.

### *Evaluation of the character set*

Design work to date indicates that most digital data processing equipment can be implemented using only the ten characters. Gate counts run higher than equipment configured from discrete IC's, with 140 percent of the IC gate count representing an approximate upper bound. Speeds appear to be comparable to the latest airborne development computers, and promise to be competitive with ground equipment as well.

For all systems where maintainability is a factor, units constructed from the character set have the obvious advantage that only ten types of spares are needed to insure system repairability. Nine of the characters are identical in all applications. The tenth, the micromemory, stores a unique program for each application. To bypass the requirement for spare ROM's of specific patterns, research is currently under way at Hughes to develop an electrically alterable ROM. The MM characters could be delivered "blank" from the manufacturer to be written into by the user with a one-shot process.

Reliability of character-built LSI computers will be enhanced by the reduction in the number of lead-bonds. Beyond that, the most significant reliability factor probably will be the type of LSI technology chosen. Bipolar TTL is a candidate for the character set mechanization due to its speed and drive capabilities. MOS is also being considered for its high packaging densities and simplicity of manufacture. Use of either or both technologies is possible depending on system requirements.

LSI enjoys a natural advantage in the diagnostic field. The arrays establish replaceable units which are quite large, thus minimizing the degree of fault isolation required. The character set in particular has several features beneficial to diagnostic procedures. The bussed structure provides several convenient points for application and observation of diagnostic signals. Also, there are only a certain number of allowable ways to inter-connect characters. This, plus the fact that there is no intervening logic, precludes the possibility of unexpected timing or logic problems arising. Once the fault detection and isolation problems are solved relative to a character, the solution is applicable to all combinations in which that character is found.

Furthermore, since every character is under the control of some micromemory, a third major approach, along with more traditional hardware and software approaches, to diagnostics becomes available. Investigations have shown that microprogram techniques are extremely effective in both detecting and isolating faults in the characters. This approach also promises fast diagnostic speeds. Not only are the diagnostics carried out at micro-instruction speeds rather than machine-instruction speeds, but in large machines each micromemory can simultaneously diagnose the characters under its control.

As an example, consider the application of these techniques to the diagnosis of the NASA MCB. Each of nine micromemories can simultaneously diagnose seven to 38 characters each. Any fault need be isolatable to one of only 206 characters, for which a replacement is chosen, assuming an operator is present, from ten basic part types. (Of course, the NASA MCB actually reconfigures automatically in case of error.)

Problems currently under investigation are diagnosis of the micromemory itself, amount and type of hardware required, and the applicability of more conventional techniques. Goals include the development of techniques for 100 percent fault detection and isolation to the character level.

The area of application stressed for the character set was computer implementation. Though the computer makes a meaningful application, there is, however, great economical advantage to be gained through application of the characters to digital equipment of unique or low volume design. Using the character methodology in such systems can reduce by large factors the engineering costs, design, and checkout time involved. To effectively achieve such a goal several design aids are desirable—a character assembler, a microprogram assembler, and a system simulator. These three programs would allow for complete design automation capability.

The character assembler input would consist of encoded instructions having the information content of a block diagram as exemplified by Figure 3. This information in conjunction with the character characteristics (which form the data base of the assembler) is processed by the assembler to produce an output consisting of wiring information for the interconnection of the characters. The character assembler output may be in the form, for example, of a wire list, an N/C tape for automatic wiring machine, or a tape input

TABLE I—Estimated MCB execution times

| INSTRUCTION | TIME IN $\mu$-Sec |
|---|---|
| FLOATING POINT ADD/SUBTRACT | $5.4 + \Delta$ |
| FLOATING POINT MULTIPLY | $21.8 + \Delta$ |
| FLOATING POINT DIVIDE | $21.0 + \Delta$ |
| STORE (MAIN MEMORY) | 7.0 |
| LOAD (MAIN MEMORY) | 7.7 |
| CONDITIONAL BRANCH | 3.8 |
| ALL SHIFTS (REGARDLESS OF LENGTH) | 5.8 |
| OR/AND | 6.2 |
| DIRECT ADD | 4.6 |
| ADD/SUBTRACT | 6.8 |
| MULTIPLY | 20.8 |

$\Delta$ = (EQUALIZATION + NORMALIZATION TIME) < 5.2 $\mu$-SEC FOR 32 BITS.

to a routing program for printed circuit card etch layout.

The encoding information for the micromemory array is provided on tape by the microprogram assembler. This tape is used directly in the manufacture or alteration of the array. The microprogram code is assembled with the usual aids provided by machine language assemblers.

System simulation would be accomplished from (1) information of the machine structure as input to the character assembler, (2) the microprogram code as input to the microprogram assembler and (3) instructions from the system designer input directly to the system simulator. The degree to which system checkout would be accomplished would of course be dependent upon the sophistication of the simulator. However, because of the high level of definition of the characters themselves the simulator would not be concerned with details of the Boolean logic or signal interface consistency between characters. Therefore a worthwhile simulator is seen as a feasible task.

Thus, the complete system—microprogrammable characters, character assembler, microprogram assembler, and system simulator—provide the system designer the capability for total system design from his desk. Furthermore, he is not concerned with logic design in any form. When he specifies the following:

1. character configuration
2. microprograms
3. simulation instruction

these item are provided for:

1. character assembly
2. back panel wiring
3. micro-array encoding
4. system checkout

all without the services of a logic designer or the technician's help. In fact, it is conceivable that no human intervention need take place between the system designer and his designed hardware!

## ACKNOWLEDGMENT

The research reported in this paper was sponsored in part by the Electronics Research Center under Contract NAS 12-665.

## REFERENCES

1 R C JENNINGS
*Design and fabrication of a general purpose airborne computer using LSI arrays*
Digest 1968 IEEE Computer Group Conf June 1968

2 N CSERHALMI O LOWENSCHUSS B SCHEFF
*Efficient partitioning for the batch—fabricated fourth generation computer*
Proc FJCC 1968

3 J J PARISER
*Connection considerations with a veiw toward batch fabrication*
Proc of the Nat Symposium of the Impact of Batch Fabrication on Future Computers April 1965 213

4 H R BEELITZ S LEVY R J LINDHARDT H S MILLER
*System architecture for large scale integration*
Proc FJCC 1967

5 J J PARISER H E MAURER
*Implementation of the NASA modular computer with LSI functional characters*
Proc FJCC 1969

# Fault location in cellular arrays*

by K. J. THURBER

*Honeywell Systems and Research Center*
St. Paul, Minnesota

## INTRODUCTION

Testing of complex integrated cellular logic circuits fabricated using LSI techniques has become a source of concern to users and manufacturers. Since an economically feasible solution to testing problems is not visible for the complex arrays contemplated for the near future, manufacturers have acknowledged the seriousness of the problem. Currently some observers believe that LSI cannot be tested because general procedures for testing and diagnosing digital circuits are applicable to small networks of approximately 30 gates, while cellular arrays are contemplated as containing hundreds or thousands of gates on one chip. However, if arrays are constrained to be in a cellular form, then testing problems can be simplified and test schedules can be produced which use the interconnection structure of cellular arrays.

In some cases the iterative interconnection structure of cellular arrays enables derivation of test schedules that exhibit an iterative nature, thus reducing the complexity of the testing problem in comparison with testing problems encountered in testing a noniterative structure containing an equal number of gates. It has been shown that the structure of single-rail cascades can be used to great advantage in the derivation of test algorithms for cascades [6] and that this testing can be accomplished from the edge of the cascade. These results are extendable to a large class of arrays. However, Kautz[1,2] has shown that cellular arrays exist which cannot be tested from their edge terminals.

* The author was formerly with the Electrical Engineering Department, Montana State University, Bozeman, Montana. This work has been supported by a National Science Foundation Grant, No. GJ-158, a National Defense Education Act Title IV Fellowship, No. 67-06596, and an Air Force Cambridge Research Laboratories Contract, No. F19628-67-C-0293.

### *Problem definition*

The iterative interconnection structure of cellular arrays allows decomposition of testing problems for LSI cellular arrays into several subproblems. One subproblem is the testing of single-rail cascades, such as the one shown in Figure 1. These cascades can be used in the production of more-complex cellular arrays, and techniques can be derived such that if a single-rail cascade can be tested then certain complex arrays can be tested. Examination of problems encountered during solution of the problem of testing single-rail cascades using only input and output terminals of cascades produces methods that can be used to test more-complex arrays. Specifically, the solution of problems involved in testing single-rail cascades lends insight to methods useful in testing cellular arrays from their edge terminals by computers using an average of only two or three tests per cell contained in the array.

Figure 2 indicates the construction of an important class of cellular arrays. An example of an important class of arrays that has this interconnection structure is a cutpoint array.[4] This array consists of collector rows and vertical cascades. Busses extend across all collector rows and distribute every variable across the vertical cascades. This construction reduces the testing of this array to the testing of a single-rail cascade, since each collector row can be tested as a single-rail cascade (under the added assumption that both a 0 and a 1 can be placed on the input to each buss that extends across the collector rows) and each vertical cascade can be tested as a single-rail cascade. Output values of vertical

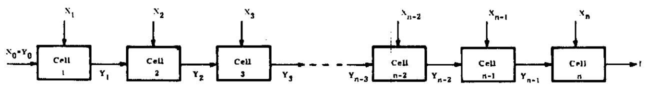


Figure 1—Interconnection structure of cascades

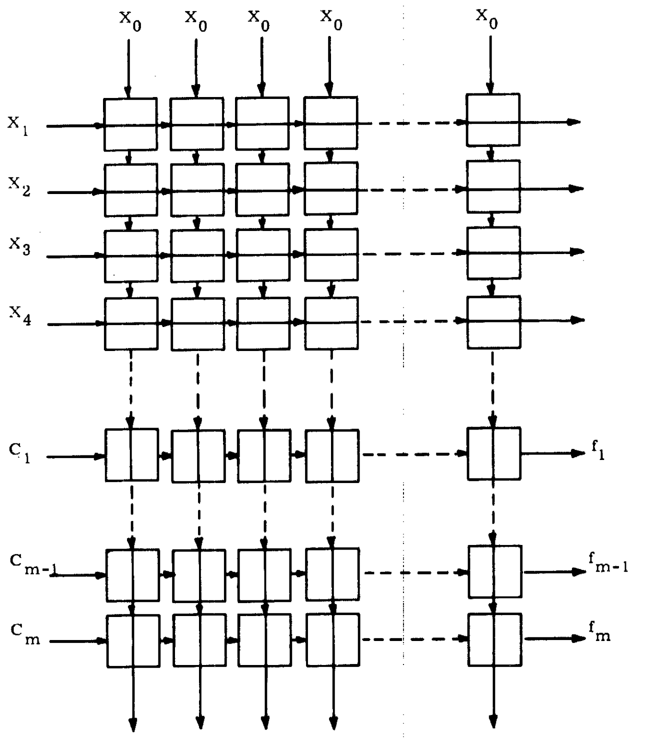


Figure 2—Construction of a testable cellular array

cascades are measured at the bottom of the array whereas collector row output values are measured on the right-hand side of the array. Admittedly, it would be desirable to test all collector rows (and all vertical cascades) simultaneously; however, to accomplish this, a restriction on the array structure must be made that restricts the class of testable arrays until the procedure becomes practically useless.

### Practical considerations

Consideration of testing problems produced by LSI chips may help develop test algorithms that could be used to test today's complex printed circuit boards. However, complex cellular arrays in practice will be more difficult to test than printed circuit boards. Consider that not only must exact error locations be indicated, but that a decision must be made based on the number of errors and their locations as to what can be done with imperfect arrays. Are imperfect arrays discarded or can they be salvaged in some manner? Minnick[5] and Spandorfer[8] have suggested that extra vertical cascades and collector rows be installed at predetermined intervals in arrays, such as in Figure 2. If a vertical cascade or collector row has an error, then the extra cascade or row could be used to produce the correct function.

Before any test procedures can be established, an error or circuit failure criterion must be established which allows definition of possible error types that may appear in LSI construction. In a later section an expanded allowable set of errors for certain types of cellular arrays will be presented.

Placing an accessible test pad on an interconnection between cells reduces the effective area usable for the cells. For this reason attempts should be made to accomplish all testing and location of faulty cells from the terminals of the array without any test pads being included in the array.

A test schedule could verify the complete truth table, transfer function, or state table for any given device; however, this procedure would require too much time and would add greatly to the expense of the array. Instead of a complete verification procedure, another solution could be to test certain input conditions on a probabalistic or expected utilization basis; however, this method is still very unsatisfactory. A feasible approach is to decide on a dominant failure mode from which a set of allowable errors can be derived for each cell type used in arrays under consideration. With this knowledge manufacturers could construct arrays using certain interconnection structures and could design cells with redundant properties. This would cause an increase in the probability that, if a failure occurs which is one of the dominant failure types, the cell error that occurs is a cell error that is contained in the set of allowable errors.

### Generation of tests and test equipment

Redundant design, failure modes, allowable errors, and required confidence level contribute to the determination of the number of tests required; however, the array's structure can almost determine the number of tests independently of these factors. Test schedules are constructed to verify whether each cell is producing its specified function. This method of testing was chosen in preference to verifying an array's truth table because the number of tests needed is generally much less than $m(2^{n+1})$, where m functions of $n + 1$ variables are produced. Under certain assumptions, choosing test schedules capable of accomplishing the task of locating every error in arrays such as shown in Figure 2 is plausible (see Theorem 1), and these test schedules can be programmed for testing using digital computers. Because of their iterative structures, cellular arrays simplify problems encountered in the detection and location of faults.

Since test schedules can be programmed for single-rail cascades, computers will be able to test many types of arrays with very minor software input changes. In particular, for the single-rail cascade under the assumptions of Theorem 1, a general fault detection program could be written. To test a cascade the only needed input information would be the cell types and their location in the cascade. With this information the general program is able to test all cascades of one type. When the type of cascade changes, this information can be given the computer as input data and all cascades of the new type can then be tested. Because of the structural interconnection of arrays shown in Figure 2, no reprogramming of the computer is needed when a new type of array appears.

## *Assumptions and definitions*

Figure 1 illustrates the interconnection structure of a Maitra cascade.[3] Every cell in the cascade is a two-input, one output cell. It is assumed that the Boolean variables applied to the cascade are numbered as illustrated on the cascade shown in Figure 1. All testing of the cascade is accomplished using only the input leads and the output lead of each cascade (and of arrays). The ability to measure the functional value produced by a cell by means of probing a buss connecting two adjacent cells is not assumed. To minimize the "uncertainties" (the functional values between cells cannot be measured and the location of the error is unknown; therefore, the functional values between cells are uncertain) involved in testing cascades, it is assumed that cell $n$ is tested first (see Figure 1), then cell $n$–1, etc. If an error occurs in cell $n$–$j$, its propagation may be stopped by one of cells $n$–1, $n$–2, $\cdots$, $n$–$j$ + 1. Once cell $n$ is tested, it may be set such that it transmits the output of cell $n$–1 to the output terminal of the cascade. In this manner (under certain error assumptions) the cells may be tested in the following order until error location results: $n$, $n$–1, $\cdots$, 1. The number of tests needed to test a cellular cascade is $0(n)$*, where $n$ is the number of cells in the cascade.

It is assumed that only one error (faulty cell) may appear in a cascade. Also, the interconnections between cells do not fail, the error is time independent; i.e., if cell $m$ is in error at time $t_1$, then cell $m$ is still in error at $t_2 > t_1$ and the error type in cell $m$ has not changed. Further, the input and output leads of the cascade do not fail.

* See Definition 6.

It is assumed that the 12 allowable cell functions for a Maitra cascade are $f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}, f_{11}, f_{13}$, and $f_{14}$. (See Definition 1 for an explanation of the notation $f_i$.) Seven allowable errors are assumed for each cell; these are $f_{15}$ (s–a–1; stuck-at-one), $f_0$ (s–a–0; stuck-at-zero), $f_{15-p}$ (complementation where $p$ is the cell function), $f_{12}$ (the input $X$), $f_3$ (the complement of the input $X$), $f_{10}$ (the input $Y$), and $f_5$ (the complement of the input $Y$). These seven errors consist of the two failure types (s–a–0 and s–a–1) usually assumed by most fault diagnosticians augmented by $f_{15-p}, f_{12}, f_3, f_{10}$, and $f_5$. [Note that $f_{10}$ and $f_5$ have different allowable error sets; i.e., $E_{f_{10}} = (f_0, f_{15}, f_5, f_{12}, f_3)$ and $E_{f_5} = (f_0, f_{15}, f_{10}, f_3, f_{12})$.]

Definition 1. The cell functions are numbered as follows:

| $X_i$ | $Y_{i-1}$ | $f_0$ | $f_1$ | $f_2$ | $f_3$ | $f_4$ | $f_5$ | $f_6$ | $f_7$ | $f_8$ | $f_9$ | $f_{10}$ | $f_{11}$ | $f_{12}$ | $f_{13}$ | $f_{14}$ | $f_{15}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Definiton 2. An *error* occurs in a cell whenever the cell produces a function that is not the same as the function specified for that cell.

Definition 3. $G = (f_1, f_2, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}, f_{11}, f_{13}, f_{14})$.

Definition 4. $I_p$ denotes (1, 2, 3, 4, $\cdots$ p).

Definition 5. The error function E is a mapping from $G$ x $I_n$ to $G$, where $E(f_i, j) = f_k$ denotes that cell j was theoretically to produce $f_i \epsilon G$ but instead it produced $f_k \epsilon G$. Clearly, $E(f_i, j) = f_i$ indicates that cell j does not have an error occurring in it.

Definition 6. X* means either X or X′, but not both.

Definition 7. O(n) means the same order of magnitude as $n$.

## *A necessary and sufficient condition for fault location in cascades*

Location of a single fault in a cascade is considered in this section. A necessary and sufficient condition for location of a single fault in a cascade is proven. The proof of Theorem 1 can be utilized to obtain an algorithm to locate faults in a cellular cascade or array.

Theorem 1. Given a cascade with $n$ cells, then the error can be located if and only if for every $i \epsilon I_n - (1)$

(1) $E(f_{14}, i) \neq f_{15}, f_{12}$
(2) $E(f_{11}, i) \neq f_{15}, f_3$
(3) $E(f_8, i) \neq f_0, f_{12}$
(4) $E(f_2, i) \neq f_0, f_3$
(5) $E(f_6, i) \neq f_9, f_{12}, f_3$
(6) $E(f_9, i) \neq f_6, f_{12}, f_3$
(7) $E(f_{13}, i) \neq f_{12}, f_{15}$
(8) $E(f_7, i) \neq f_3, f_{15}$
(9) $E(f_4, i) \neq f_0, f_{12}$
(10) $E(f_1, i) \neq f_0, f_3$
(11) $E(f_{10}, i) \neq f_0, f_{15}, f_5$
(12) $E(f_5, i) \neq f_{10}, f_0, f_{15}$

Proof: The proof is an induction proof. Clearly, the theorem is true for the case $n = 1$. Assume that the theorem is true for a positive integer $k$ and consider a cascade with $k + 1$ cells. Given the cell function for cell $k + 1$, if it can be shown that the error can be located in cell $k + 1$ if and only if assumptions (1) through (12) are valid for cell $k + 1$, then the proof is complete.

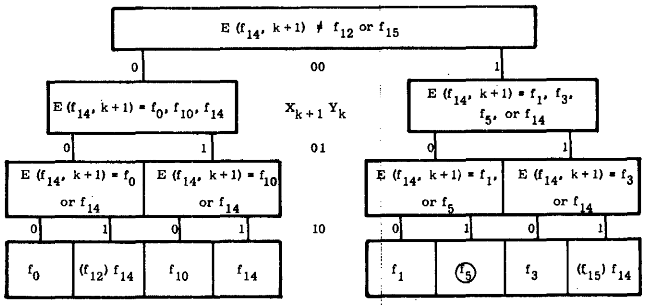


Figure 3—Test decision map for $f_{14}$

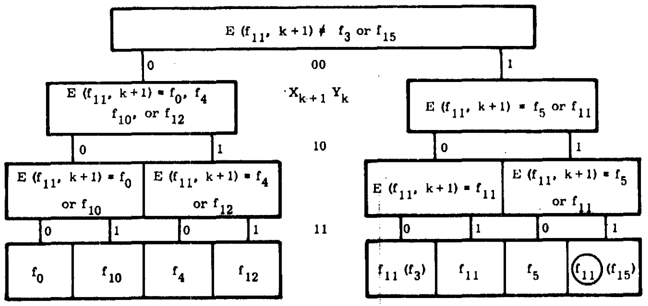


Figure 4—Test decision map for $f_{11}$

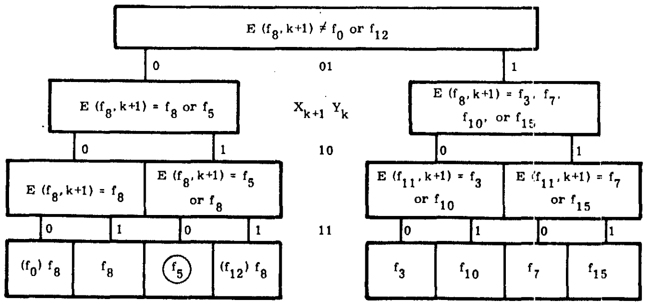


Figure 5—Test decision map for $f_8$

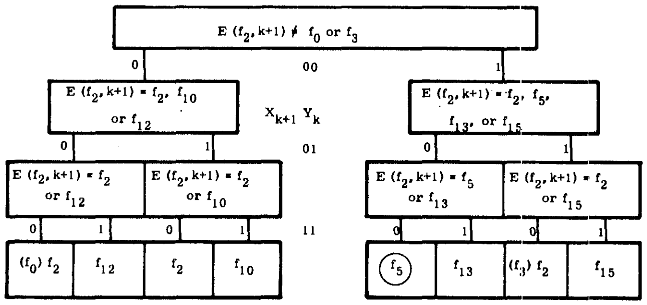


Figure 6—Test decision map for $f_2$

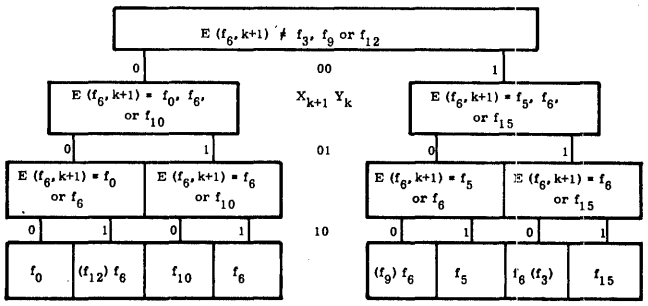


Figure 7—Test decision map for $f_6$

Assume conditions (1) through (12). This part of the proof is now completed in Figures 3 through 14. Note that if $C_0, C_1, \cdots, C_i$ are used to set $Y_i = C$ at time $t_1$, then if $Y_i = C$ is wanted at time $t_2$ if $C_0, C_1, \cdots, C_i$ are utilized again, $Y_i$ is the same value as it was at $t_1$; however all that can be said about $Y_i$ is that it is either $C$ or $C'$, but not both. This fact is used in the proof of this theorem. In the figures with the circled function number it may be necessary to add one more test to deter-

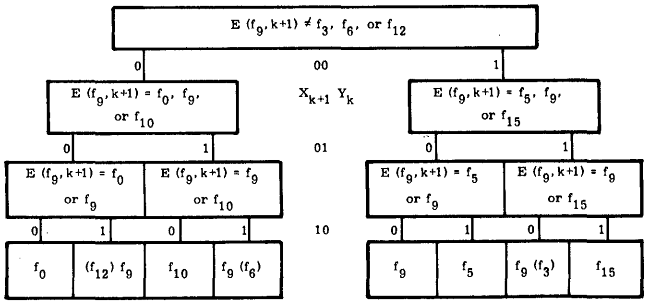


Figure 8—Test decision map for $f_9$

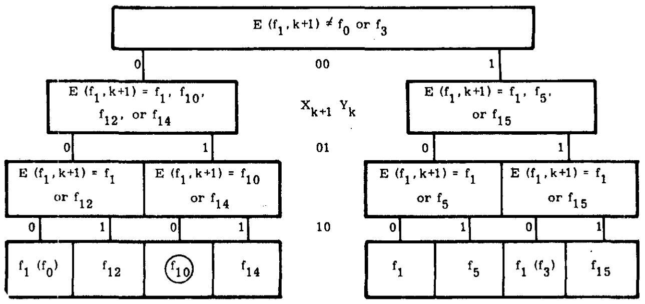


Figure 12—Test decision map for $f_1$

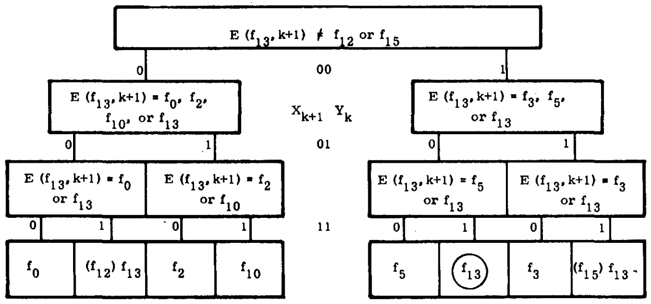


Figure 9—Test decision map for $f_{13}$

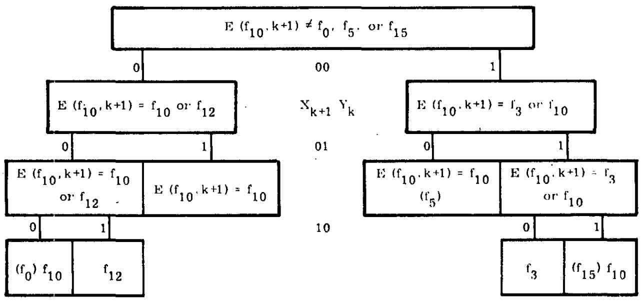


Figure 13—Test decision map for $f_{10}$

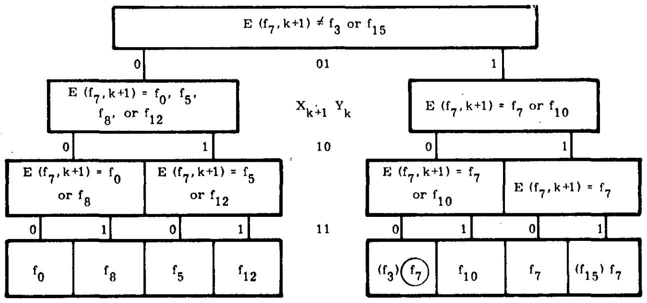


Figure 10—Test decision map for $f_7$

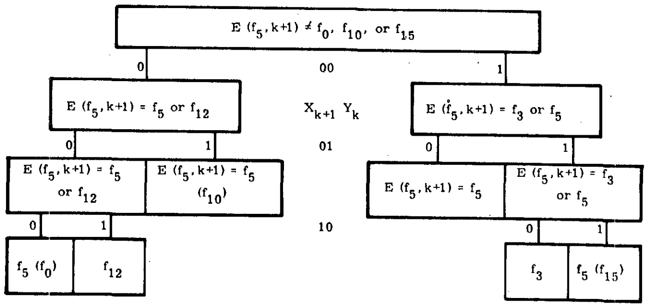


Figure 14—Test decision map for $f_5$

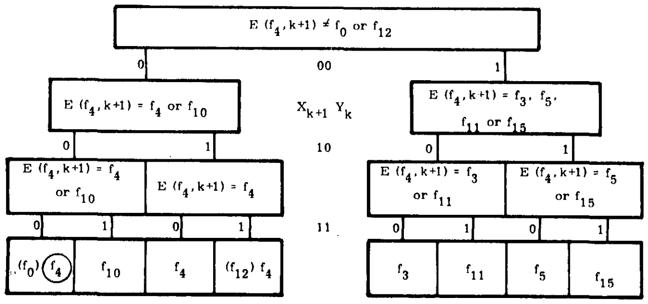


Figure 11—Test decision map for $f_4$

mine whether the cell is in error or is receiving the complemented sequence.

The proof of the other half of the theorem will be by contradiction. Assume that the error can be located, but that the restrictions (1) through (12) are not needed. Then it can be verified that the following pairs of conditions give the same output at the cascade's terminal. Since the two conditions give the same outputs, the error cannot be located, which is a contradiction of the assumption; therefore, the assumption that the restrictions are not needed is incorrect and the proof is completed. After (1) an abbreviated notation is used. Note: Using the Test Decision Maps and the contradiction part of this proof one can actually determine the values of $Y_{i-1}$.

(1) $Y_k = 1, 1, 1$ and $E(f_{14}, k + 1) = f_{14}$ are equivalent to $Y_k = 0, 1, 0$ and $E(f_{14}, k + 1) = f_{15}$ at the cascade's output terminal.

$Y_k = 0, 0, 0$ and $E(f_{14}, k + 1) = f_{14}$ are equivalent to $Y_k = 0, 1, 0$ and $E(f_{14}, k + 1) = f_{12}$ at the cascade's output terminal.

(2) $Y_k = 0, 0, 0$ and $E(f_{11}, k + 1) = f_{11}$;
$Y_k = 0, 0, 1$ and $E(f_{11}, k + 1) = f_3$.
$Y_k = 1, 1, 1$ and $E(f_{11}, k + 1) = f_{11}$;
$Y_k = 0, 0, 1$ and $E(f_{11}, k + 1) = f_{15}$.

(3) $Y_k = 1, 1, 1$ and $E(f_8, k + 1) = f_8$;
$Y_k = 1, 0, 1$ and $E(f_8, k + 1) = f_{12}$.
$Y_k = 0, 0, 0$ and $E(f_8, k + 1) = f_8$;
$Y_k = 1, 0, 1$ and $E(f_8, k + 1) = f_0$.

(4) $Y_k = 1, 1, 1$ and $E(f_2, k + 1) = f_2$;
$Y_k = 0, 1, 1$ and $E(f_2, k + 1) = f_3$.
$Y_k = 0, 0, 0$ and $E(f_2, k + 1) = f_2$;
$Y_k = 0, 1, 1$ and $E(f_2, k + 1) = f_0$.

(5) $Y_k = 1, 1, 1$ and $E(f_6, k + 1) = f_6$;
$Y_k = 0, 1, 0$ and $E(f_6, k + 1) = f_3$.
$Y_k = 0, 0, 0$ and $E(f_6, k + 1) = f_6$;
$Y_k = 0, 1, 0$ and $E(f_6, k + 1) = f_{12}$.
$Y_k = 1, 0, 1$ and $E(f_6, k + 1) = f_6$;
$Y_k = 0, 1, 0$ and $E(f_6, k + 1) = f_9$.

(6) $Y_k = 1, 1, 1$ and $E(f_9, k + 1) = f_9$;
$Y_k = 0, 1, 0$ and $E(f_9, k + 1) = f_{12}$.
$Y_k = 0, 0, 0$ and $E(f_9, k + 1) = f_9$;
$Y_k = 0, 1, 0$ and $E(f_9, k + 1) = f_3$.
$Y_k = 1, 0, 1$ and $E(f_9, k + 1) = f_9$;
$Y_k = 0, 1, 0$ and $E(f_9, k + 1) = f_6$.

(7) $Y_k = 1, 1, 1$ and $E(f_{13}, k + 1) = f_{13}$;
$Y_k = 0, 1, 1$ and $E(f_{13}, k + 1) = f_{12}$.
$Y_k = 0, 0, 0$ and $E(f_{13}, k + 1) = f_{13}$;
$Y_k = 0, 1, 1$ and $E(f_{13}, k + 1) = f_{15}$.

(8) $Y_k = 1, 1, 1$ and $E(f_7, k + 1) = f_7$;
$Y_k = 1, 0, 1$ and $E(f_7, k + 1) = f_3$.
$Y_k = 0, 0, 0$ and $E(f_7, k + 1) = f_7$;
$Y_k = 1, 0, 1$ and $E(f_7, k + 1) = f_{15}$.

(9) $Y_k = 1, 1, 1$ and $E(f_4, k + 1) = f_4$;
$Y_k = 0, 0, 1$ and $E(f_4, k + 1) = f_0$.
$Y_k = 0, 0, 0$ and $E(f_4, k + 1) = f_4$;
$Y_k = 0, 0, 1$ and $E(f_4, k + 1) = f_{12}$.

(10) $Y_k = 1, 1, 1$ and $E(f_1, k + 1) = f_1$;
$Y_k = 0, 1, 0$ and $E(f_1, k + 1) = f_0$.
$Y_k = 0, 0, 0$ and $E(f_1, k + 1) = f_1$;
$Y_k = 0, 1, 0$ and $E(f_1, k + 1) = f_3$.

(11) $Y_k = 1, 1, 1$ and $E(f_{10}, k + 1) = f_{10}$;
$Y_k = 0, 1, 0$ and $E(f_{10}, k + 1) = f_{15}$.
$Y_k = 0, 0, 0$ and $E(f_{10}, k + 1) = f_{10}$;
$Y_k = 0, 1, 0$ and $E(f_{10}, k + 1) = f_0$.
$Y_k = 1, 0, 1$ and $E(f_{10}, k + 1) = f_{10}$;
$Y_k = 0, 1, 0$ and $E(f_{10}, k + 1) = f_5$.

(12) $Y_k = 1, 1, 1$ and $E(f_5, k + 1) = f_5$;
$Y_k = 0, 1, 0$ and $E(f_5, k + 1) = f_0$.
$Y_k = 0, 0, 0$ and $E(f_5, k + 1) = f_5$;
$Y_k = 0, 1, 0$ and $E(f_5, k + 1) = f_{15}$.
$Y_k = 1, 0, 1$ and $E(f_5, k + 1) = f_5$;
$Y_k = 0, 1, 0$ and $E(f_5, k + 1) = f_{10}$.

If the cascade meets the assumptions of Theorem 1, then Theorem 1 can be used to determine test schedules for the location of an error in cascades. It should be noted that when cell $k$ is tested, one obtains information about the cells $k - 1, k - 2, \cdots, 1$, and therefore a test schedule with $0(n)$ tests will test any cascade with $n$ cells under the allowable error set[6]. Clearly, if the conditions of Theorem 1 are relaxed, then fault detection (and maybe isolation) can be accomplished in the same number of tests; however, if one is only interested in fault detection, Theorem 2 is the best technique to use.

If a more complex cascade than the cascades considered here is under consideration, then a good understanding of the method used to derive the theorems in this paper will allow one to extend the theories presented. If the cell functions $f_0, f_3, f_{12}$, and $f_{15}$ are allowed, then the fault techniques may be easily extended since none of these functions depend on the $Y$ value; however, one must exercise care in the use of the theory because it is based on the ability of the tester to place theoretically both a 0 and a 1 on the $Y$ interconnection, and examples (trivial) in which this cannot be accomplished do exist.

## *Fault detection in Maitra cascades*

In this section the detection of a single fault in a cascade is considered. The theory for this section is based on the observation that every $n$ cell Maitra cascade (as defined in this paper) produces a function dependent on $X_0^6$.

The purpose of this detection scheme is to utilize exactly two tests to detect whether a cascade has a faulty cell.

Theorem 2. Let the Maitra cascade have $n$ cells. If $C_1$ $C_2, \cdots, C_n$ are such that $f(X_0, C_1, C_2, \cdots, C_n) = X_0{}^*$, then

(1) $f(1, C_1, \cdots, C_n) = f(0, C_1, \cdots, C_n)$ implies that there exists a cell i such that $E(f_p, i) = f_0, f_{15}, f_{12}$, or $f_3$.

(2) $f(1, C_1, \cdots, C_n) = (1^*)'$ and $f(0, C_1, \cdots, C_n) = (0^*)'$ imply that there exists a cell $i$ such that $E(f_p, i) = f_{15-p}$ or $f_5$.

(3) $f(1, C_1, \cdots, C_n) = 1^*$ and $f(0, C_1, \cdots, C_n) = 0^*$ imply that there is no error in the cascade or that there exists a cell such that $E(f_p, i) = f_{10}$ and $p \neq 10$.

Proof: In part (1) $f$ does not depend on $X_0$; therefore, there must be a cell $i$ such that $E(f_p, i) = f_0, f_{15}, f_{12}$, or $f_3$. In part (2) $f$ depends on $(X_0{}^*)'$; therefore, there is a cell $i$ such that $E(f_p, i) = f_{15-p}$ or $f_5$. Whereas, the proof of part (3) is now obvious.

$X_0$ was chosen as the variable to be used in Theorem 2 because of the symmetry of the resulting theorem. Since $X_1$ can be made (by a suitable choice of constants, to pass theoretically through every cell*, the theorem could be rewritten in terms of $X_1$. In terms of the complexity of the detection scheme it is seen that cascades could have a very simple detection test schedule. It should be noted that Theorem 2 can very easily be adapted to provide fault detection in cascades if it is assumed that $f_{10}$ is not an allowable error for any of the 12 cell functions.

*Examples*

This section consists of examples of the use of Theorems 1 and 2. $f_A$ denotes the measured value of $f$ whereas $f_T$ denotes the theoretical value of $f$.

* Assuming the cell function for cell 1 is not $f_{10}$ or $f_5$.

Example 1. Assume that there is no error in the cascade shown in Figure 15.

Test

| $X_0$ | $X_1$ | $X_2$ | $X_3$ | $X_4$ | $f_T$ | $f_A$ | Conclusion |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 | |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | $E(f_6, 4) = f_6$ |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | $E(f_8, 3) = f_8$ |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | $E(f_{14}, 2) = f_{14}$ |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | $E(f_{14}, 1) = f_{14}$ |

Example 2. Assume that $E(f_8, 3) = f_{15}$ in the cascade shown in Figure 15.

Test

| $X_0$ | $X_1$ | $X_2$ | $X_3$ | $X_4$ | $f_T$ | $f_A$ | Conclusion |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 | |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | $E(f_6, 4) = f_6$ |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | $E(f_8, 3) = f_{15}$ |

Example 3. Assume that $E(f_{14}, 2) = f_3$ in the cascade shown in Figure 15.

Test

| $X_0$ | $X_1$ | $X_2$ | $X_3$ | $X_4$ | $f_T$ | $f_A$ | Conclusion |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | $E(f_6, 4) = f_6$ |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | $E(f_8, 3) = f_5$ so an extra test is needed. |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | $E(f_8, 3) \neq f_5$ and the complemented sequence $Y_2$ is being received. |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | $E(f_{14}, 2) = f_3$ |

Example 4. This example satisfies the hypothesis of Theorem 2. Assume that $E(f_8, 4) = f_0$ for the cascade shown in Figure 15.

$$[(X_0 + X_1 + X_2) X_3] \oplus X_4 = f_T(X_0, X_1, X_2, X_3, X_4)$$

$$f_T(X_0, 0, 0, 1, 0) = X_0$$

$f_A(0, 0, 0, 1, 0) = f_T(1, 0, 0, 1, 0) = 0$ implies that there is a cell $i$ such that $E(f_p, i) = f_0, f_{15}, f_{12}$, or $f_3$.

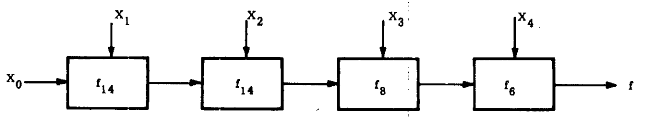


Figure 15—A cascade to be tested

## CONCLUSION

Techniques for fault location and detection in cellular arrays with an allowable error set of $f_0$, $f_{15}$, $f_{15-p}$, $f_3$, $f_{12}$, $f_5$, or $f_{10}$ were described in this paper. It was shown that the problem of testing an array could be reduced to the problem of testing a cascade. The solutions presented are particularly attractive because of their simplicity. To locate an error, $0(n)$ tests are needed for an $n$ cell cascade. Detection of an error requires only two tests if the allowable error set is reduced by one error ($f_{10}$).

A necessary and sufficient condition for single-error location was given. If the restrictions of this condition are relaxed, then an isolation theorem such as given by Thurber [6,7] can be derived; however, this isolation condition will be more complex than the theorem given by Thurber [6,7]. A criterion that enables detection of a single error in only two tests was derived.

Although the theories presented were derived for regular arrays of logic, they have potentially wide areas of application. A good understanding of the philosophies presented here will allow the extension of the results to cascades of $m$ input $n$ output cells. Also, some irregular arrays may be tested using this theory if they can be decomposed into sections composed of some form of a cascaded structure (or sections composed of structures closely resembling a cascaded structure).

## ACKNOWLEDGMENT

The author wishes to thank R. C. Minnick for his help in the preparation of this paper.

## REFERENCES

1 W H KAUTZ
*Testing for faults in combinational cellular logic arrays*
1967 Switching and Automata Theory Symposium

2 W H KAUTZ
*Diagnosis and testing of cellular arrays, properties of cellular arrays for logic and storage*
SRI Project 5876 Scientific Rpt No 3 July 1967 119-145

3 K K MAITRA
*Cascaded switching networks of two-input flexible cells*
IRE Trans on Electronic Computers Vol EC-11 April 1962 136-143

4 R C MINNICK
*Cutpoint cellular logic*
IEEE Trans on Electronic Computers Vol EC-13 Dec 1964 685-698

5 R C MINNICK
*A survey of microcellular research*
Journal Association for Computing Machinery Vol 14 April 1967 203-241

6 K J THURBER
*Fault location in cellular arrays*
PhD dissertation Montana State Univ June 1969

7 K J THURBER
*Fault location in cellular cascades*
Submitted to IEEE Trans on Computers

8 L M SPANDORFER J V MURPHY
*Synthesis of logic functions on an array of integrated circuits*
Scientific Rpt No 1 for UNIVAC Project 4645 AFCRL-63-528 Contract AF 19(628)2907 Sperry Rand Corp UNIVAC Engineering Center Oct 1963

# Fast multiplication cellular arrays for LSI implementation

*by* C. V. RAMAMOORTHY and
S. C. ECONOMIDES

*The University of Texas at Austin*
Austin, Texas

## INTRODUCTION

The inherent capabilities of Large Scale Integration technology have recently shifted attention toward two major concepts in the design of functional computer subsystems; the concepts of Functional Modules and Cellular Arrays.

The Functional Module concept emphasizes the possible standardization of frequently used common digital subsystem units such as registers, adders, counters, etc. Because of the unique iterative properties also displayed by these units it is common to view them as building blocks (functional modules), built on a single substrate of material, the interconnection of which can expand significantly their functional capabilities. In addition to standardization, their massive production may suggest low cost subsystems.

The Cellular Array concept allows the interconnection of several types of mutually independent logic blocks, the cells, in various geometric configurations to perform a desired operation.

This paper is an attempt to combine the above two approaches in the realization of a Binary Cellular Array multiplication unit easily adaptable to the LSI realization techniques and speculate the possibilities of the realization of other similar such functional units aiming to lower the cost per unit of computation and possibly increase the overall system reliability.

Multiplication was chosen in the study because it forms the basis of division and square root operations by iterative methods as well as others indicated by design trend of present day computing systems.

The methodology and retroactive design procedures of the Multiplication Array are presented. Interconnection arrangements at the cell level, for the array formation, as well as the module level by bringing all module inputs and outputs at the terminals of the "package", for the purpose of assembling larger multiplication units, are also shown.

Since in any LSI circuit testing imposes a complex problem some diagnostic schemes are suggested for reconfiguration and operation under reduced capabilities or even by automatically switching in of a permanently connected spare module.

Other LSI considerations in terms of cell or module fan-in/fan-out, total number of pins required per package, chip sizes and densities and rough cost estimates are also discussed.

### *Single bit multiplier*

Figures 1 and 2 show the integral parts and the detailed cellular array structure of the multiplication unit, in which each row of the array corresponds to one bit of the multiplier. The array uses K-bit operands producing 2K bit product.

To achieve fast execution time the multiplication is done by performing K-1 carry save additions (simple EXCLUSIVE-OR operations) followed by a full binary addition. Since the cells in the array operate asynchronously, the unit as a whole can operate faster without using a clock pulse.

We shall next explain the single-bit multiplication unit in some detail.

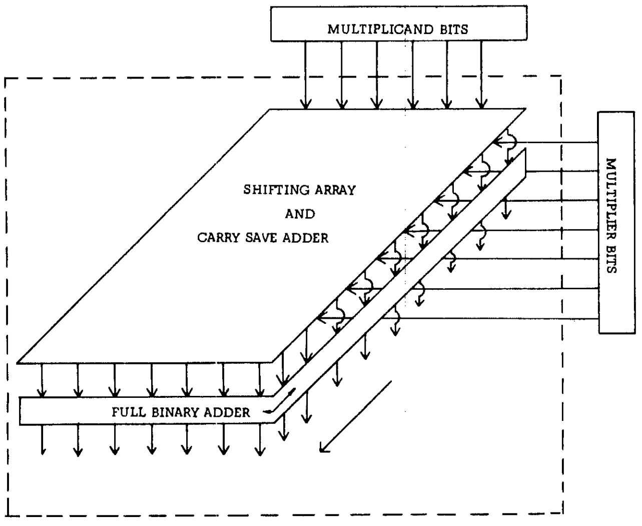


Figure 1—The integral parts of the asynchronous multiplication array

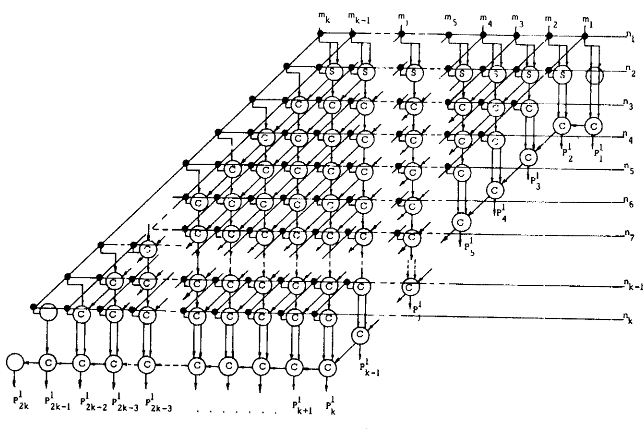


Figure 2—The "single-bit" asynchronous multiplication cellular array

Let the multiplicand be represented by the binary vector $M = (m_1, m_2, \cdots m_k)$ and the multiplier by the binary vector $N = (n_1, n_2, \cdots n_k)$.

A kx2k, P matrix is now generated starting from right to left (whose elements $p_{ij}$ are computed from the relation $p_{ij} = m_i \cdot n_j$, $p_{ij} \epsilon \{0, 1\}$ with the following conditions

$$p_{ij} = \begin{cases} m_{j-i+1} \text{ if } n_i = 1 \text{ and/or } \begin{cases} 1 \leq i \leq k \text{ for } i = 1, 2, 3, \ldots k \\ i - 1 < j < k + 1 \text{ for } i = 1, 2, 3 \ldots k \end{cases} \\ 0 \text{ if } n_i = 0 \text{ and/or } \begin{cases} 1 \leq i \leq k \text{ for } i - 1, 2, 3 \ldots k \\ k + 1 \leq j \leq i - 1 \text{ for } i = 1, 2, 3 \ldots k \end{cases} \end{cases}$$

In terms of the array to be implemented, this condition implies that for the range "i," "j" where $p_{ij} = 0$ no cell will be required to perform a logic function. Thus the [P] matrix has the following form:

$$\begin{array}{cccccc} p_{1,2k-1} & p_{1,2k-2} \ldots p_{1,k} \ldots p_{13} & & & p_{12} & p_{11} \\ p_{2,2k-1} & p_{2,2k-2} \cdots p_{2,k} \cdots p_{23} & & & p_{22} & p_{21} \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ p_{k,2k-1} & p_{k,2k-2} & p_{k,k} & p_{k,3} & p_{k,2} & p_{k,1} \end{array}$$

The following example will illustrate the above matrix formation.

EXAMPLE

——————— $M = (10101)$ and $N = (111111)$

MULTIPLY...

then the P matrix is $P =$

$$\begin{array}{ccccccccc} 0&0&0&0&1&0&1&0&1 \\ 0&0&0&1&0&1&0&1&0 \\ 0&0&1&0&1&0&1&0&0 \\ 0&1&0&1&0&1&0&0&0 \\ 1&0&1&0&1&0&0&0&0 \end{array}$$

The above matrix can be realized by selective AND-ing of components of $M$ and $N$. This "Shifting Network" accomplishes the proper positioning of the numbers to be added before their addition, just as in the conventional multiplication. Arrays of Carry Save Adders are used to perform the addition of these binary numbers utilizing Wallace's algorithm.[1]

The first stage of the Carry Save Adder adds the first two rows of the P matrix (first two generated partial products) thus generating two vectors—the first partial sums and the first carry having the form:

$$s = (s_{1,\,2k-1} \quad s_{1,\,2k-2} \ldots s_{1,\,k} \ldots s_{11})$$

$$c = (c_{1,\,2k-1} \quad c_{1,\,2k-2} \ldots c_{1,\,k} \ldots c_{11});\ s_{ij},\ c_{ij} \epsilon \{0, 1\}$$

The double subscript is used to identify the above vectors with corresponding positions of the P matrix that contributes to their generation.

The logic functions yielding the elements $s_{2j}$ and $c_{2j}$ are:

$$s_{2j} = p_{1j}\overline{p}_{2j} + \overline{p}_{1j}\,p_{2j}$$

$$c_{2j} = p_{1j}\cdot p_{2j}$$

where j = 2, 3,···2k − 1. The composite cells are shown in Figure 3a.

In the subsequent stages the Carry Save Adder will add three vectors: The sum vector generated at the previous stage, the carry vector generated at the previous stage shifted once to the left and the next row vector of the P matrix.

The logic functions producing the new s and c vectors

$$s = (s_{i,\,2k-1}\,s_{i,2k-2}\ldots s_{i,\,k}\ldots s_{i1})$$

$$c = (c_{i,\,2k-1}\,c_{i,\,2k-2}\ldots c_{i,\,k}\ldots c_{i1})$$

for i = 3, 4,. . .k, and j = 1, 2, 3. . .2k − 1 are:

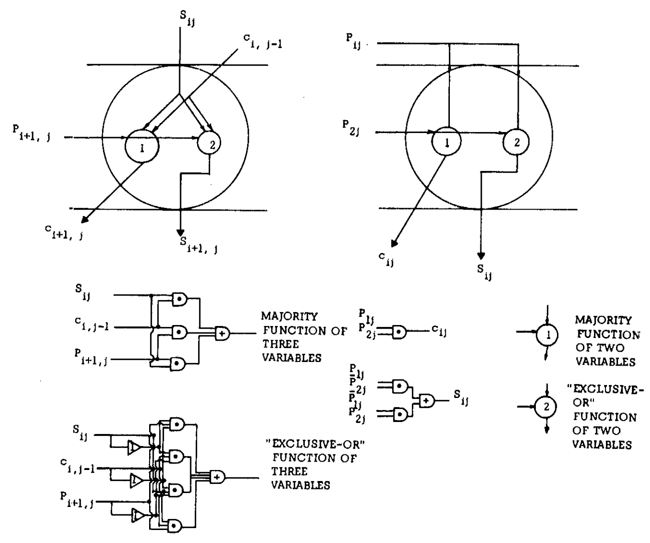
$$c_{i+1,j} = s_{ij}c_{i,j-1} + s_{ij}p_{i+1,j} + c_{i,j-1}p_{i+1,j}$$

$$s_{i+1,j} = \overline{s}_{ij}\overline{p}_{i+1,j}c_{i,j-1} + \overline{s}_{ij}p_{i+1,j}\overline{c}_{i,j-1} + $$
$$+ s_{ij}\overline{p}_{i+1,j}\overline{c}_{i,j-1} + s_{ij}p_{i+1,j-1}$$

The composite cell 'C' is shown on Figure 3b. After the Carry Save Addition has been performed for all the partial product row vectors of the matrix P, a Ripple Binary Adder is used to add the sum and carry row vectors of the last stage of the Carry Save Adder. The typical cell of this Ripple Binary Adder has the same structure as "cell C" of the Carry Save Adder, except that it ripples through the carries generated to next high order position and puts out the correct binary sum which of course involves any carry incident into it from the previous stage. The output of the Ripple Binary Adder is the final product of the multiplication.

The superposition of the "Shifting Array", the "Carry Save Adder" and the "Ripple Binary Adder" resulting in the "Single Bit Multiplication Cellular Array" is as shown in Figures 1 and 2.

It was found that with a Carry Save Adder there is considerable gain in the time propagation over the choice of Full Binary Adder. Assuming a uniform delay d for each cell in the array, the total execution time $T_p$ of k bit by k bit multiplication is bounded between the limits (k-1) d $\leq T_p \leq$ 2kd. The lower limit (k-1) d is the total delay in the Carry Save Adder while the



$s_{ij}$
$c_{i,j-1}$
$P_{i+1,j}$
$c_{i+1,j}$
$s_{i+1,j}$
$P_{1j}$
$P_{2j}$
$c_{ij}$
$s_{ij}$
1
2
MAJORITY FUNCTION OF THREE VARIABLES
"EXCLUSIVE-OR" FUNCTION OF THREE VARIABLES
MAJORITY FUNCTION OF TWO VARIABLES
"EXCLUSIVE-OR" FUNCTION OF TWO VARIABLES

Figure 3a, b—Cell "S", Cell "C"

upper limit 2kd depends on the choice of the device for the final Full Binary Addition. This, as compared to the maximum delay requirement in a conventional multiplier due to k(k-1) d full binary additions plus k-single bit left shifts. The asynchronous multiplication array, as implemented is shown in Figure 2.

*Two-bit multiplier*

Upon examination of this array it was decided that the time propagation and therefore the computational speed could be further improved by reducing the Carry Save Adder stages, in other words, the rows of the array. This also improves the attenuation factor of the cell inputs as they ripple throug the array.

An alternate multiplying algorithm, examining the multiplier bits in subsets of two, was investigated resulting in the block diagram of Figure 4 which displays the integral parts of the modified array. To illustrate the algorithm better, this was assumed to be an m × n instead of a square array and the multiplier parts now are:

1. The m + n + 2—bit register for the multiplicand
2. The n + 2—bit register for the multiplier
3. The m + n + 3—bit registers for the final product
4. The Binary Shifting Array (BSA)
5. The Input Control Circuit (ICC)
6. The Carry Save Adder (CSA)
7. The "End Around Carry" Accumulator (EACA).

Before investigating the above circuits the general algorithm concept must be established. This algorithm

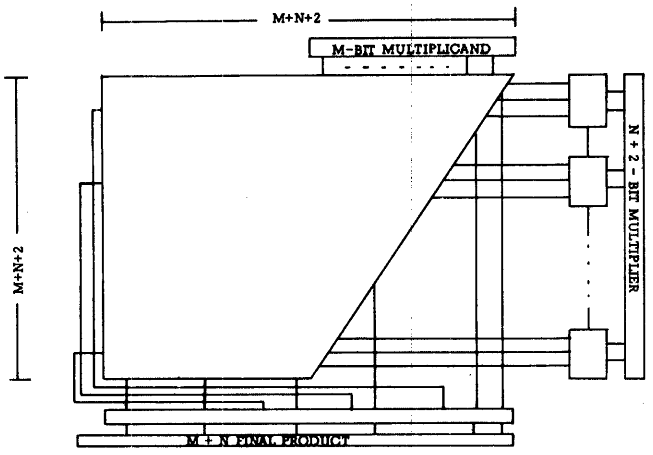


Figure 4—The modified "two bit" multiplier

calls for three types of decisions in each multiplication stage: ADD or SUBTRACT a single multiple of the multiplicand and SHIFT without generating any multiples of the multiplicand. This is opposed to the conventional multiplication which requires only shifts of the multiplicand and their addition. For the possible four 2-bit combination one has the following obvious interpretation:

a. Combination $00_2 = 0_{10}$ Add nothing to the partial product
b. Combination $01_2 = 1_{10}$ Add one times the multiplicand to the partial product
c. Combination $10_2 = 2_{10}$ Add two times the multiplicand to the partial product
d. Combination $11_2 = 3_{10}$ Add three times the multiplicand to the partial product.

Combinations (a), (b) impose no difficulty in their generation. Combination (c) requires a 1-bit shift of the multiplicand to the left according to the obvious simple fact: to generate any $2^n$-th multiple of a binary number (the multiplicand in this case), where n is any integer $n \geq 0$, shift the number n-bit position to the left. For example, to generate $16 \times m = 2^4 \times m$, shift m four bit positions to the left. For combination (d) one notices that the multiplicand can be expressed in the following two ways:

$$(1)\quad (4 \times m) - (1 \times m) \qquad (2)\quad (2 \times m) + (1 \times m).$$

The first representation was chosen for this multiplication algorithm, according to which a complementation (2's complement) of one times the multiplicand is performed and added to the corresponding present stage of the multiplication array while a request is issued to add one times the multiplicand in the following stage *in that order.* The latter request is taken care of by adding "1" to the bit pair of the multiplier corresponding to the next multiplication stage, thus increasing the pair's integer value by one. This is commonly known as a "carryout."

The subtraction of multiplicand from the partial product is performed in two stages. The one's complement of m is "added" into the Carry Save Adder of the row. A "one" in the lowest order bit position corresponding to the row is generated and inserted into the End-Around-Carry Accumulator (EACA), at the appropriate column. Together this constitutes adding the two's complement of m after appropriate shifting. Thus any sequential borrow propagation is prevented at the Carry Save Adder stages. Since the "End-Around-one's" if generated by any or all rows are inserted at distinct columns of the EACA the latter performs at most one accumulation during a complete multiplication cycle. It must be remembered that partial products generated at each row are bussed to the next row of cells with a 2-bit left shift.

The following two tables indicate the decisions that have to be made when the various bit pair combinations are encountered at a given stage when no carryout (Table I) or a carryout (Table II) has been generated in the previous stage.[5]

| *Bits* | *Multiple of m generated* | *Carryout* |
|---|---|---|
| 00 | 0 | 0 |
| 01 | 1 | 1 |
| 10 | 2 | 2 |
| 11 | −1 | −1 |

Table I

| *Bits* | *Multiple of m generated* | *Carryout* |
|---|---|---|
| 00 | 1 | 0 |
| 01 | 2 | 0 |
| 10 | −1 | 1 |
| 11 | 0 | 1 |

Table II

Finally to illustrate the overall performance of the modified multiplier with minimum effort an example of a 4-bit positive multiplicand times a 6-bit positive multiplier producing a $6 + 4 = 10$-bit long product is presented. The extension of the algorithm and techniques involved can be easily extended for an arbitrary bit length multiplicand or multiplier.

## The binary shifting array

The BSA generates the elements $p_{ij} \epsilon (0,1)$ and such that

$$p_{ij} = 0 \text{ for } 1 \leq j \leq m + 1; i \geq 3$$
$$j \leq m + n; i \geq 3$$

where

$$m = \text{no. of bits in } M$$
$$n = \text{no. of bits in } N$$

with the rest of the $p_{ij}$'s varying according to the corresponding multiplicand bits. Its implementation procedure is as follows:

1. Provide for one additional bit pair at the most significant part of the multiplier by inserting two zeroes in the register. This will take care of a possible generation of a "carryout" at the two most significant bits of the multiplier. Provide for as many zeros to the left-hand side of the multiplicand register to make it $(m + n + 2)$ bits long.
2. Examine the multiplier bits two at a time from the least significant to the most significant bits.
3. Generate the following three numbers for each multiplier bit pair:
   a. The multiplicand
   b. The multiplicand inverted (one's comple-complement)
   c. Twice the multiplicand.

Repeat for the next bit pair until all n-multiplier bits are used. For this particular example the procedure will yield the formation of possible CSA inputs, where the "boxed in" numbers will be the rows of the P matrix chosen by the control lines of the ICC (Figure 5a).

The two numbers are placed in the registers with the least significant bit of the multiplier starting at the top. For every bit pair of the multiplier there is a corresponding triplet of "AND" gate rows and one of inverters, all together being capable of generating any of the desired forms of the multiplicand called for in Tables I and II.

The "AND" gates have two inputs and one output, one of the inputs being a multiplicand bit bussed across and the other being the appropriate line activated by the ICC. The outputs of the leftmost column are used to keep count of the "End Around Carries" and are directly connected to the appropriate positions of the EACA.

## The input control circuit

The ICC is a column of $(n/2 + 2)$ rectangular cells (see Figure 4). Its operation is to select the appropriate multiplicand multiple for each possible bit pair combination, by the way of three output lines: $L_1$, $L_2$, $L_3$.

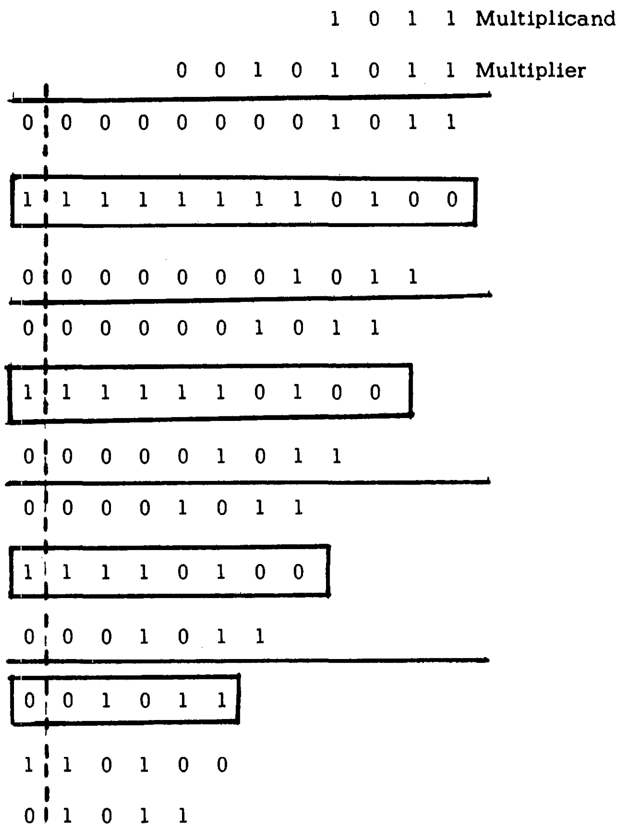


Figure 5a—Multiplication example

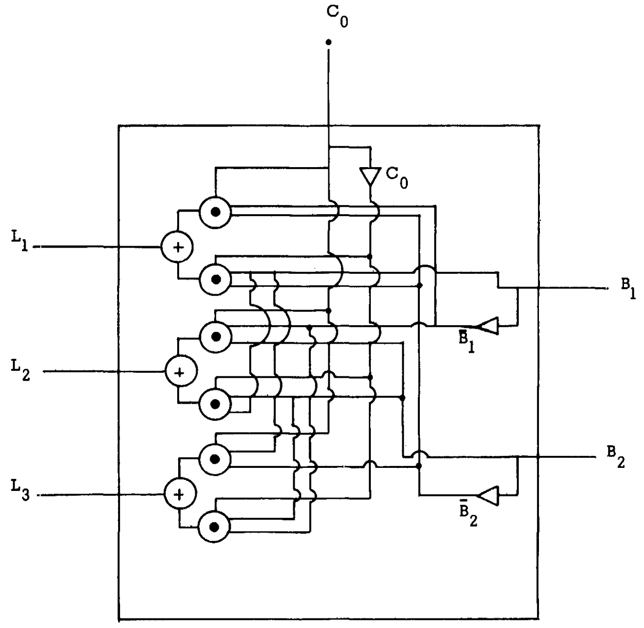


Figure 5b—Cell-K of the input control circuit

$L_1$ activates the single multiple of the multiplicand (first "AND" gate row of each group of rows in the ESA). $L_2$ activates the 2's complement of the multiplicand (second "AND" gate row, directly under each row of inverters). $L_3$ activates the double multiple of the multiplicand. Therefore, the typical cell of the ICC has $B_1$, $B_2$, and $C_0$ as inputs and $L_1$, $L_2$ and $L_3$ as outputs. Its logic functions are shown below. $B_1$ and $B_2$ are any two consecutive bits and $C_0$ is the carryout. The logic:

|  |  |  | M | $\overline{M}$ | 2M |
|---|---|---|---|---|---|
| $B_1$ | $B_2$ | $C_0$ | $L_1$ | $L_2$ | $L_3$ |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

Note: The interpretation of $B_1, B_2 = 01$ is not one times the multiplier as it would obviously appear, but it is instead two times the multiplicand because of the way the multiplier is placed in the register, vertically with the least significant bit on the top. The $B_1$, $B_2$ = 10 combination is interpreted in a similar manner

$$L_1 = \overline{B}_1\overline{B}_2C_0 + B_1\overline{B}_2\overline{C}_0$$

$$L_2 = \overline{B}_1B_2C_0 + B_1B_2\overline{C}_0$$

$$L_3 = B_1\overline{B}_2C_0 + \overline{B}_1B_2\overline{C}_0$$

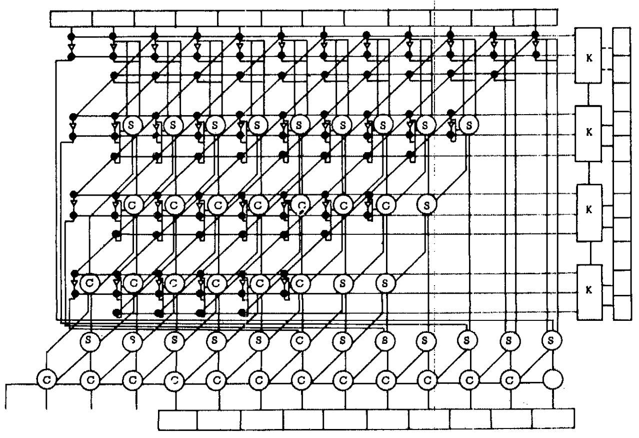


Figure 6—The binary multiplying cellular array

The typical cell "K" of the ICC is shown in detail in Figure 5b.

### The carry save adder, end around carry accumulator and full binary adder

A layout of the inputs to the CSA stages, the EACA and FBA is displayed below. The groups of binary numbers between the lines represent the actual inputs to a particular row of cells. The first three groups are CSA row inputs. The fourth group represents the EACA inputs and the final group, those of the FBA. All binary numbers representing partial products are of course $P$ matrix row vectors activated by the ICC lines due to a particular multiplier bit pair combination.

```
    1 1 1 1 1 1 1 0 1 0 0   1st partial product
    1 1 1 1 1 0 1 0 0       2nd partial product
  -----------------------
    0 0 0 0 0 1 0 0 1 0 0   1st partial sum
  1 1 1 1 1 0 1 0 0 0 0 0   1st carry
    1 1 1 0 1 0 0           3rd partial product
  -----------------------
  1 0 0 0 1 1 0 0 0 1 0 0   2nd partial sum
0 1 1 1 0 0 1 0 0 0 0 0 0   2nd carry
  0 1 0 1 1                 4th partial product
  -----------------------
  0 1 0 0 0 1 0 0 0 1 0 0   3rd partial sum
1 0 1 0 1 1 0 0 0 0 0 0 0   3rd carry
              0   1   1   1 End Around Carries
-------------------------
1 0 0 0 1 1 1 0 1 0 0 0 1   4th partial sum
0 1 0 0 0 0 0 0 0 1 0 0 0   4th carry
-------------------------
1 1 0 0 1 1 1 0 1 1 0 0 1   Final Sum (Result)
```

Figure 6 shows array after superimposing the individual circuits.

It can be easily noticed that there is a reduction by a factor of two in the total number of cell rows required for the array and therefore in the total final propagation $T_p$, at the expense of some additional control logic, a number of inverters and an additional stage for the EACA. No further complexity in the cell structure results, thus the originally developed cells were used, with a minor modification for cell S as shown in Figure 7a. This cell may also be present in the single bit multiplication array.

It must also be noticed that the overflow of bits resulting in the left-most significant part of the final

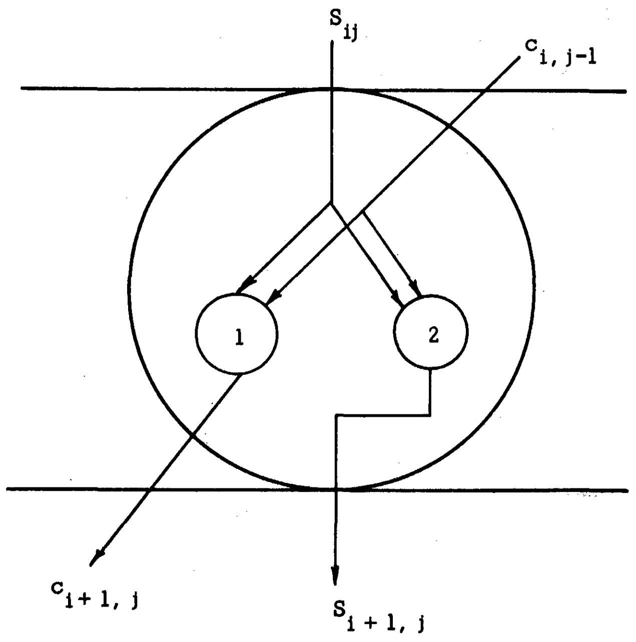


Figure 7a—Cell "S"—A form of Cell "S"

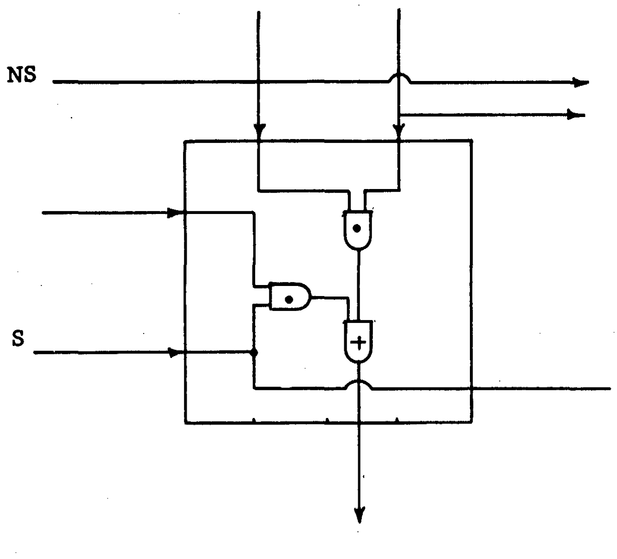


Figure 7b—Cell "R"—Reconfiguration cell

product register may be advantageously utilized for sign and decimal point considerations.

## *Diagnostics and reconfiguration*

In order to incorporate diagnostics in the array and study the interconnection problem, a standard size module had to be assumed. It was felt that the implementation of a 64 × 64 bit multiplier would be a good choice for all practical purposes. An interconnecting scheme of standard dimension 64 × 8 bit modules to realize the 64 bit multiplier was then devised aiming to minimize the number of pins per module necessary for the interconnection.

As seen in Figure 8, the resulting 64 × 64 multiplication unit requires 2-Full Binary addition stages and 4-Carry Save addition stages per module, a total of 32-Carry Save additions and 15-Binary Additions (only one for the first module). However, there is a real time overlap between these various stages, and by utilizing a pipelining technique and a series of flip-flops after each FBA, a 100 percent utilization of the unit during computation is achieved, and the multiplication cycle is considerably faster. This is illustrated shortly in connection with Table III.

The basic module as displayed in Figure 6 has to be modified further for the interconnection. An extra FBA and additional gating for diagnostic purposes is

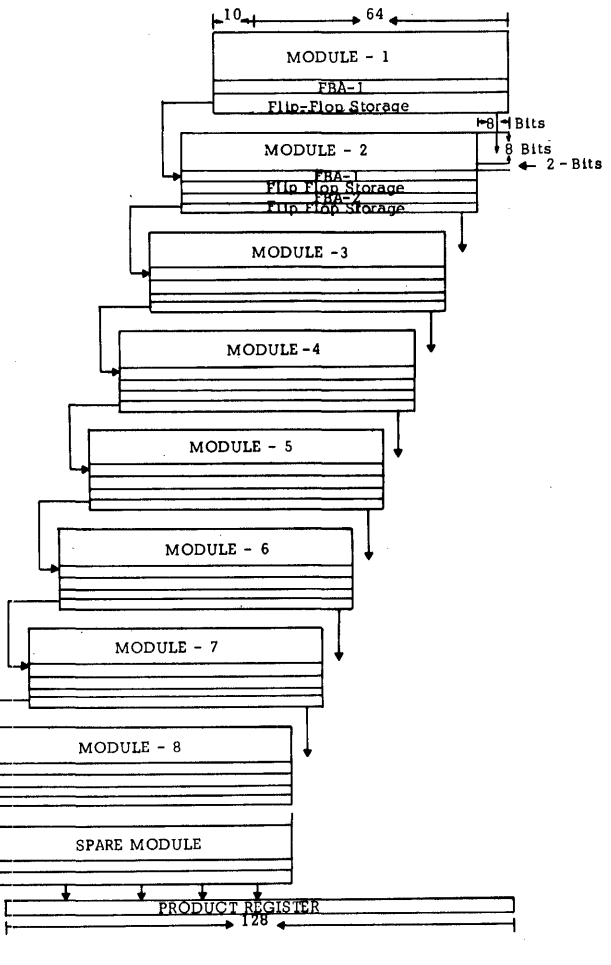


Figure 8—Example of an assembled 64 × 64-bit multiplication unit using the pipelining scheme

introduced in every module between the output of its respective FBA and what is shown as a product register. The typical newly developed cell for the diagnostics and reconfiguration is shown in Figure 7b, while the above mentioned modifications are displayed in detail in Figure 9 for a typical module.

As seen, three additional control lines are needed to perform the following functions.

a. To relay a Fault or No-Fault signal, indicating that a fault has or has not occurred in one particular module (NF/F) (e.g., if $F = 0$ $NF = 1$).
b. To relay a No Shift signal for the output of this module, ($NS = 1$) if no fault has occurred in the preceding module.
c. To relay a shift, eight-bits to the right, ($S = 1$) for the output of this and all subsequent modules if a fault has been detected in the preceding module.

The detection of the fault could be accomplished by a software routine which may check the final product of the unit periodically and appropriately set the flip-flops of the control signals.

By shifting the outputs of all subsequent modules to the malfunctioning one eight-bit positions to the right while forcing the output of the faulty module to be equal to zero at the same time and simultaneously introducing the spare module which is permanently connected to the unit, one can still achieve 100 percent computational efficiency. If another module fails to function properly, by applying again the same reconfiguration scheme the unit will function with a reduced capability since the eight-least significant bits of the multiplier will be lost. No provision has been made at this point if two modules fail to function properly

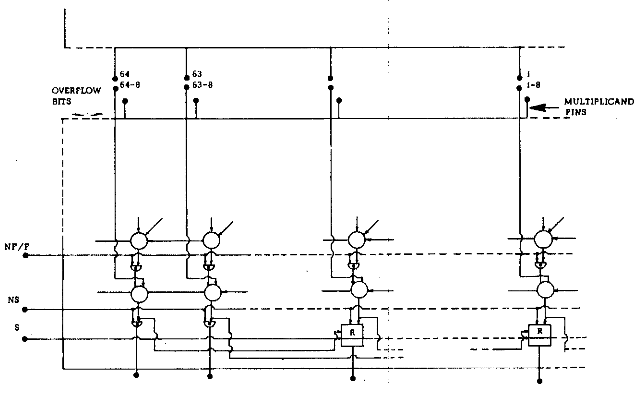


Figure 9—The combinational logic gating for reconfiguration

at the same time. At least one of them must be replaced to put the multiplication unit back in service.

Aiming to maximize the number of multiplications per unit time, as already mentioned, one can introduce storage elements at intermediate points. This allows the unit to accept a new set of operands without waiting for the total completion of the present computation.

Consider an $m \times m$ bit multiplier module. If the intermediate computations are stored after the Carry Save adders, the first Binary adder and the second Binary adder, the rate of multiplications in the module per unit time will be

$$R_m = \frac{1}{\max\,[t_{cs}, t_b]} \text{ where}$$

$t_{cs}$ = Total time propagation through the CSA.

$t_b$ = Total time propagation through the FBA for the binary addition of two m-bit binary numbers.

Then the number of storage elements required per module is $2m + m + m = 4m$. If, however, storage elements are inserted at the outputs of the two Binary Adders only, as shown in Figure 8, the maximum rate of multiplications in each module per unit time will be

$$R_{max} = \frac{1}{t_b + t_{cs}}$$

while the total number of storage elements required will be decreased by half, that is 2m.

The table below gives the sequence of events in the first four modules of the $64 \times 64$ composite multiplier unit of eight modules, based on the pipelining technique.

Table III

| | MODULES | | | |
|---|---|---|---|---|
| TIME UNITS | 1 | 2 | 3 | 4 |
| 1 | $B_{11}$ | $B_{11}$ | $B_{11}$ | $B_{11}$ |
| 2 | $B_{21}$ | $B_{21}$, $B_{12}$ | $B_{21}$ | $B_{21}$ |
| 3 | $B_{31}$ | $B_{31}$, $B_{22}$ | $B_{31}$ | $B_{13}$ |
| 4 | $B_{41}$ | $B_{41}$, $B_{32}$ | $B_{41}$, $B_{23}$ | $B_{41}$, $B_{14}$ |
| 5 | $B_{51}$ | $B_{51}$, $B_{41}$ | $B_{51}$, $B_{33}$ | $B_{51}$, $B_{24}$ |

Each time unit in the above table corresponds to the factor $t_b + t_{cs}$, and $B_{ij}$ represents the $j^{th}$ binary addition of the $i^{th}$ multiplication.

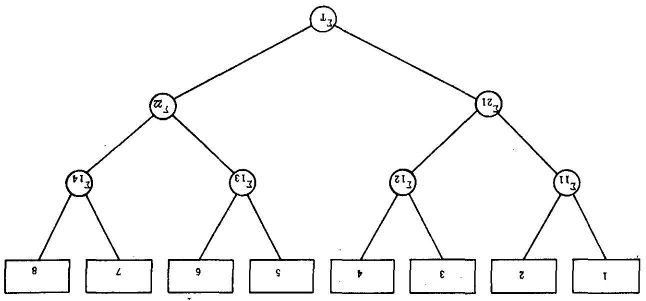


Figure 10—An alternate interconnecting scheme for the 8-modules of the 64 × 64 multiplication unit

Another interconnecting scheme which has not been investigated yet in detail but seems to be equally as efficient, considerably faster and adaptable to the proposed reconfiguration technique is the one shown in Fig. 10, where each level of nodes represents FBA's Figure 10, where each level of nodes represents FBA's performing in parallel with an anticipated multiplication cycle of

$$[1 + \log_2 n]\, t_b + t_{cs}\,.$$

*LSI implementation*

The implementation shown for the 64 × 8 module reveals a number of characteristics suitable for large scale integration. Among them are the repetitive interconnections of simple identical cells and the modularity suitable for expansion and reconfiguration.

Below some of the approximate hardware requirements are pointed out.

### Approximate number of PINS/MODULE

1. $m + n + 2$ needed for the multiplicand register
2. $m + n + 2$ needed as inputs to the second FBA
3. $m + n + 2$ needed for the product
4. $n + 2$ needed for the multiplier register
5. three-control pins for reconfiguration

### Approximate number of CELLS/MODULE

The cells are the kinds already discrussed: C, S, S′, R, K. All are present in a module.

1. $m \times n/2$ cells needed for the CSA stages
2. $m + n$ cells needed for the EACA stage
3. $m + n$ reconfiguration cells
4. $2\ (m + n + 2)$ cells needed for the two FBS's
5. $n/2 + 1$ cells needed for the ICC.

### Approximate number of GATES/CELL*

For cell "C" approximately seven-gates are required
For cell "S", "S′" approximately three-gates are required
For cell "R" approximately two-gates are required
For cell "K" approximately nine-gates are required

The above estimates point out the fact that testing at the individual cell or circuit level (item yet to be examined) becomes a problem, especially when the complexity of the chip is increased, with a paralleled decrease in reliability and yield of non-defective chips. However, using the modular approach it is advisable to perform the testing externally on the module and discard the malfunctioning units. This would considerably decrease the amount of logic on a chip, which would otherwise have to be inserted for the testing of the individual circuits. This approach seems to be economically feasible since it is estimated that by 1970 an LSI chip of 100 × 100 mils in size may contain 200 components, at five cents per component, while by 1975 an LSI chip of 300 × 300 mils in size may contain as many as 3,600 components at the cost of about one cent per component. Therefore, miniaturization of LSI chips will discourage the testing on the individual circuit level, while the loss due to the discarding of modules after tesing at the frame level, will be negligible.

In view of the above considerations and since the present state-of-art high density MOS circuits are being driven at 10 MHz, implementation of the multiplier modules as the one presented by MOS circuits appears very desirable from a manufacturing viewpoint. A reasonable building block might be a 64 × 64 bit multiplication unit requiring an approximate number of 5000 active elements (field effect transistors). One could also visualize the whole unit incorporated in one or two chips. Where speed is the primary requirement, the unit can be designed using fast bipolar transistors, with an expected five ns delay. Assuming then a 64 × 64 bit module is implemented by bipolar transistors, the execution time could be in the neighborhood of $0.225\mu s$, which when pipelined, the maximum number of multiplications per second may be approximately $5 \times 10^6$. An MOS array of the same module will perform in an order of magnitude slower than in the bipolar case.

* The above gates are mostly "AND" gates with the "OR" gate not included in the count. They are also $2(m + n)$ additional gates needed for the reconfiguration scheme and $m \times n$ gates for shifting each array.

The pin count also indicates that the current design is within the state-of-art of the MOS technology.

The performance figures given above are educated guesses since the circuit and intermodule delays are dependent on the circuit types, their interconnections, the chip topology, etc. In addition, the design examples described in the previous sections indicate the ease with which the array could be partitioned to fit reasonable unit or chip sizes.

## CONCLUSION

Since fast multiplication has become the basis of iterative divisions and square roots in fast computers[6,7] there appears to be a need for cheap array type, LSI realizable multiplication subsystems. This paper reports the design methodology and the detailed implementation of one such structure. Ease of diagnosis and capability of reconfiguration were used as twin requirements in the final design. When the unit is composed of a number of modules and a malfunction is detected in one of them, a method of switching automatically in a spare module was presented. An estimate of the logic circuitry in the hard core (that portion of the unit which must be operating without any faults) during testing is found to be less that 14 percent for a $32 \times 32$ module, 9.7 percent for $64 \times 64$ module and 4 percent for $128 \times 128$ module. Therefore, as the size of the multiplication module-unit increases the relative size of the hard core decreases very rapidly.

To conclude, the cellular array implementation of an asynchrouous multiplication unit using mostly non-carry-propagating Carry Save adders was accomplished. The final cell design and the control and the reconfiguring circuitry are quite simple.

A number of additional studies needs to be done in the future. The design of self-diagnosable and repairable functional arrays appear quite feasible and worth considering. The possibility of composite design of a multiplication, division and square rooting unit using techniques presented in this paper could be very useful, particularly if the division and square root algorithms are based on the availability of fast multiplication units such as those discussed in this paper.

## ACKNOWLEDGMENTS

The authors would like to thank Mr. Gary Wang of the NASA Electronics Research Center for sharing with them some of his thoughts on the subject. Also Mr. W. R. Adrion, graduate student at the University of Texas at Austin for his constructive suggestions.

## REFERENCES

1 C S WALLACE
*A suggestion for a fast multiplier*
IEEE Trans Prof Group on Electronic Computers Vol 13 No 1 Feb 1964

2 *Methods for high-speed addition and multiplication*
NBS Cir No 591 1958

3 O L MacSORELY
*High-speed arithmetic in binary computers*
Proc IRE Vol 49 No 1 Jan 1961

4 M LEHMAN
*Short-cut multiplication and division in automatic binary digital computers*
Proc Inst Elec Eng Paper No 2693M Vol 105B Sept 1958

5 I FLORES
*The logic of computer arithmetic*
Prentice-Hall Inc 1963

6 D FERRARI
*A division method using a parallel multiplier*
IEEE Trans Prof Group on Electronic Computers Vol 16 No 2 April 1967

7 S F ANDERSON et al
*IBM system model 91: Floating point execution unit*
IBM Journal of Research and Development Jan 1967

# The Pad Relocation technique for interconnecting LSI arrays of imperfect yield

*by* D. F. CALHOUN

*Hughes Aircraft Company*
Culver City, California

## INTRODUCTION

The interconnection of circuits required in Large Scale Integration (LSI) using multi-level metalization above monolithic semiconductor arrays is taking basically two approaches. One is predicated on processing with a reasonable yield entire arrays without any semiconductor defects (i.e., 100 percent yield chips) which allows once-generated fixed-wiring patterns to obtain the required interconnect. The second approach aims at much larger semiconductor arrays (i.e., full-slice LSI) for which defect-free processing cannot be expected. Thus, probe tests are made of the semiconductor circuits processed on each LSI slice (or wafer) and record is made of the good and bad circuit positions. Unique interconnection masks are then generated to interconnect good circuits in each wafer's particular yield pattern using certain "discretion" in avoiding the bad circuits. As a result, the 100 percent yield approach emphasizes the need to use standard interconnect masks but is complexity limited by the occurrence of defective circuits in larger arrays, whereas approaches capable of routing around the defective circuits have required a full set of unique signal interconnect masks for each wafer's particular yield pattern.

The Pad Relocation approach, however, allows the interconnection of full-slice LSI arrays containing defective circuits to be accomplished with a minimal amount of unique interconnect per array. Only a portion of one of the typically three interconnect levels varies from array to array, thus allowing significant improvements in the cost, reliability, and testability of the finished arrays as well as less limitation on cell yields and array complexities.

### *Description of the Pad Relocation technique*

Pad Relocation is a technique which allows a predetermined standard pattern of good circuits to be established on all LSI slices used to perform the same array function regardless of the varying yield patterns determined by DC wafer probe tests. This is accomplished by relocating the pads of nearby good circuits to the positions where good circuits were specified by a prescribed master pattern, but were not found during wafer probe tests. The pad positions above a bad circuit (or any unused circuit) are isolated from that circuit by a layer of dielectric. Where good circuits are found in expected good circuit locations, those circuits are used without relocation. Thus, the Pad Relocation technique functionally establishes a specified pattern of good circuits as if there had actually been a 100 percent circuit yield in that pattern. A single wiring pattern can then be generated for all the LSI arrays of the same function to accomplish the much more complex signal interconnect between the master pattern circuits. By determining standard cross-under areas within the Pad Relocation layer where relocation lines need never occur, it has been shown that large arrays can be interconnected with the same number of total interconnect layers as required by discretionary techniques.

With each wafer's good circuits located in the predetermined master pattern, an optimal standard interconnect of the circuits can be made for each wafer. Since this signal routing and mask-making expense is incurred only once for each function, much more effort can be spent optimizing the signal routing. As a result, the total number of interconnect levels (including Pad Relocation) may actually be fewer (for very complex arrays) than other techniques by which the interconnect is generated for each wafer's particular yield pattern.

The Pad Relocation technique has been 100 percent successful for all integrated circuit and special LSI wafers considered so far. The "master pattern" gives the prescribed locations of good circuits to which each LSI array's particular yield will be tailored. Statistically, if M is the percentage of wafer circuits in the master pattern and Y is the wafer circuit yield from probe tests, then only M(100 - Y)/100 percent of all wafer circuits need to be relocated. For example, if Y = 35 percent and M = 30 percent, then the relocation (as a statistical average) of 19.5 percent of the wafer circuits will establish a master pattern that uses 86 percent of all the good wafer circuits. This would allow 120 good circuits to be located in prescribed positions, leaving an average of only 20 good circuits unused.

## An example

The methodology of the Pad Relocation technique is best described by example. Figure 1 shows the mapping of circuits on an LSI wafer. Each dot represents the position of a semiconductor cell such as a full adder, or a quad two-input NAND gate cell, or a flip-flop, etc. Figure 2 identifies with a slash (/) the location of all circuits determined to be good by dc wafer probe tests on a particular slice. The yield of wafer circuits varies from 10 percent to 90 percent depending on the circuit complexity, and the locations of the good circuits cannot be predicted from wafer to wafer. This makes it impossible to use standard interconnect patterns without first transforming the various wafer yield patterns to a single standard pattern. The circuit yield (the percent of total circuits which are good) for the wafer in Figure 2 is nearly 30 percent and yet there is not a single area of 100 percent yield that is larger than three circuits by two circuits. Thus, 100 percent yield could obtain units with only about 5 percent of the complexity allowed by full-slice interconnection techniques. The goal is to tailor by some efficient means the locations of the good circuits in Figure 2 to a standard pattern that may be used for all wafers with about the same circuit yield. For higher yield wafers, there are other standard patterns which use more good circuits.

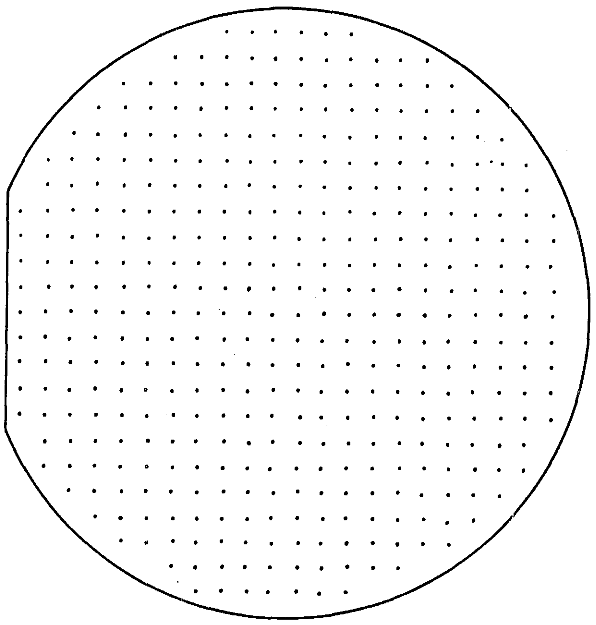


Figure 1—Integrated circuit wafer

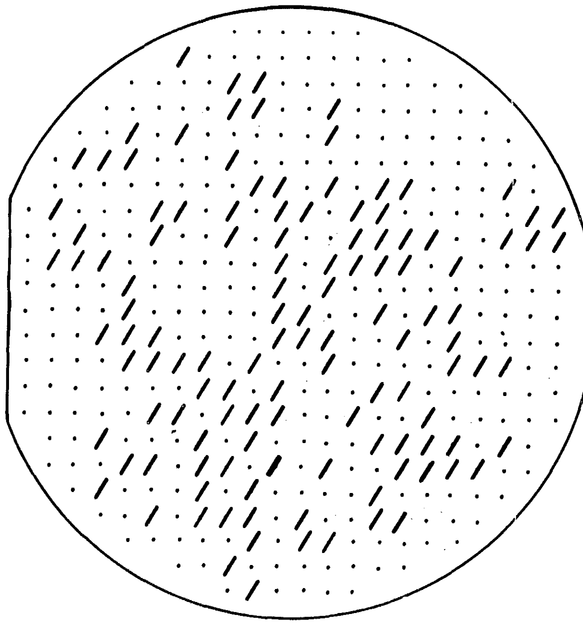


Figure 2—Wafer after test—Slashes show good circuit positions

Figure 3 shows a master pattern (in heavy dots) which can be used for wafers having at least a 25 percent yield. That pattern is characterized by a more

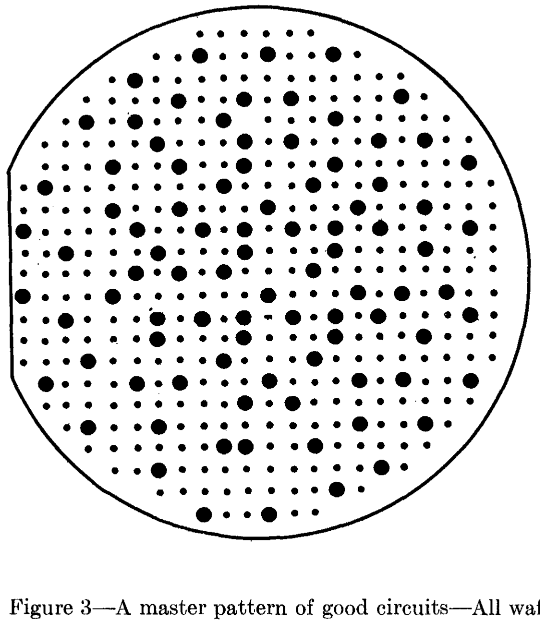


Figure 3—A master pattern of good circuits—All wafers will be matched to this pattern by the Pad Relocation technique

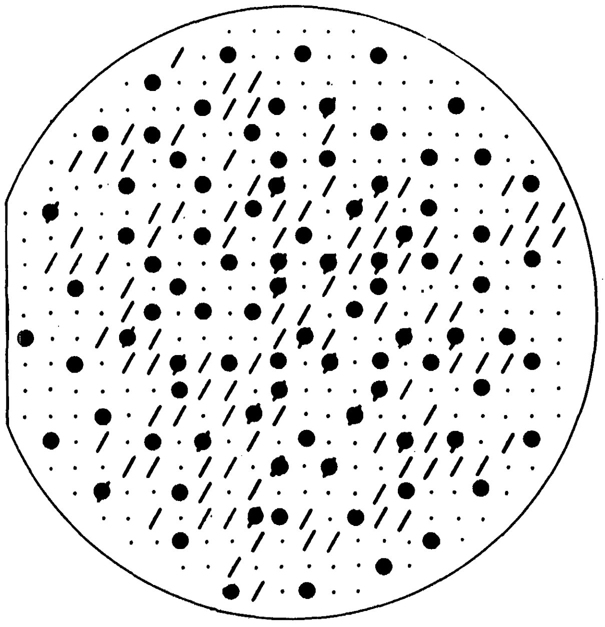


Figure 4—Master pattern superimposed on the particular yield of the Figure 2 wafer

dense usage of good circuits toward the center of the wafer with good circuit positions never adjoined on more than one side by another circuit in the master pattern. The latter characteristic facilitates the routing of standard signal interconnect as well as the relocation of circuits in at least three directions. The matching of the master pattern to the expected yield distribution as a function of distance from the wafer center optimizes the conflicting goals of minimum number of relocations and maximum probability of fulfilling the master pattern.

Figure 4 shows the Figure 3 master pattern superimposed on the particular wafer yield of Figure 2. The objective now is to route a nearby good circuit, shown by a slash, to each heavy dot (i.e., master pattern position) which initially is without a good circuit. This specification can be completed manually giving a coding sheet description of necessary circuit relocations; or a simple computer routing program can output a punched tape or cards that can be used to make a mask automatically. The computer routine for Pad Relocation will use about two orders of magnitude less run time than a customized signal routing primarily because no circuit placement or logic signal routing are required. Pad Relocation requires only that a good circuit be identified for relocation to each position in the master pattern which did not initially have a good circuit. A later paper will present work that is under way to automate the Pad Relocation selection and specification with the use of interactive graphics.

Figure 5 shows a manually generated specification

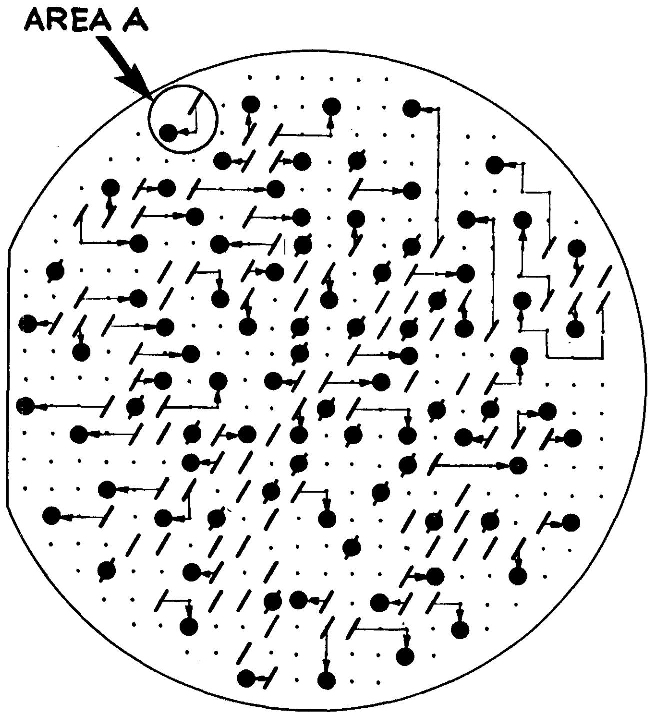


Figure 5—Specification of a set of relocations necessary to completely implement the master pattern of Figure 3

of possible relocations that completely satisfies the master pattern of Figure 3, using the good circuit positions of the wafer in Figure 2. The longest relocation line length is less than 0.45 inch. Figure 6 shows how the relocation in area A of Figure 5 can be accomplished without crossovers for a quad two-input gate cell. Each gate of the bad circuit at the lower left is functionally replaced with a good gate from the top right circuit. It should be noted that the computer needs only subroutines for leaving (or entering) a cell from the top, bottom, left, and right, for moving parallel lines over some number of cells, and for making ninety degree turns in order to do all the possible Pad Relocation routing patterns. Figure 7 shows the actual Pad Relocation of an SN5480 gated full adder above a silicon wafer using 0.002 inch aluminum lines on 0.0035 inch centers. Figure 8 shows how simple the Pad Relocation mask is if it is considered as a set of the above mentioned subroutines.

## Intermediate step to full wafer LSI

Figure 9 shows an intermediate step to full-wafer LSI using the Pad Relocation technique. Three 4-bit Modular Multiplier modules are to be fabricated from the three bordered half-inch square areas (as was suggested in a 1968 FJCC paper by D. F. Calhoun). Within the three bordered areas, slashes again represent good circuits and circles show the master pattern locations. The lines terminating in arrowheads show how three, eight, and five good circuits can be relocated into the positions circled to establish the same pattern of good circuits for each module, thus allowing the use of one standard signal interconnect pattern for all subsequent modules tailored to that pattern.

Figure 10 demonstrates the simplicity of a coding sheet specification of the necessary circuit relocations

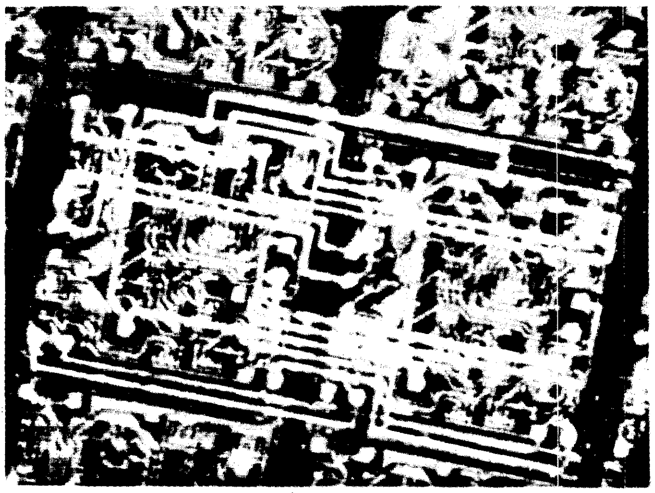


Figure 7—Pad Relocation of an SN5480 gated full adder above a silicon wafer (Using 0.002-inch aluminum lines on 0.0035-inch centers)

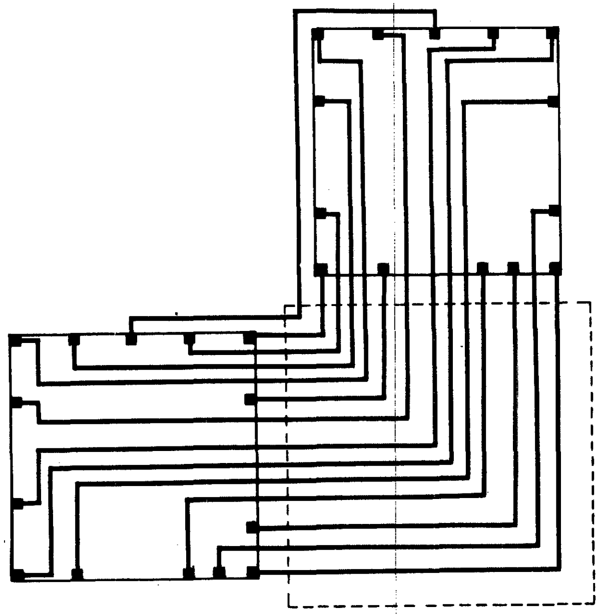


Figure 6—A set of pad relocations necessary to replace functionally the quad two-input gate circuit in area A of Figure 5

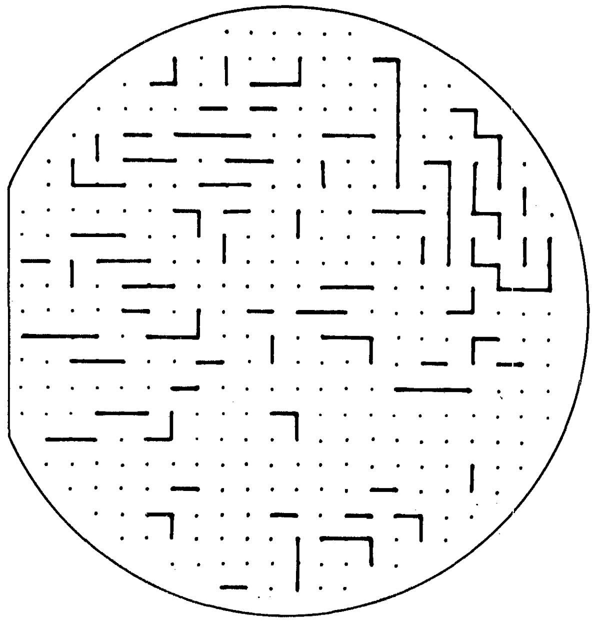


Figure 8—Mask pattern for the pad relocations specified in Figure 5

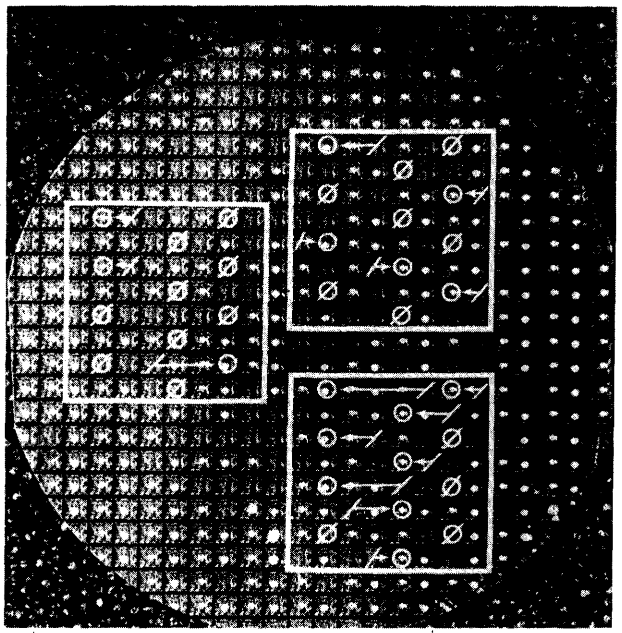


Figure 9—Pad Relocation routing for three 200-gate modules on a single 1-½inch wafer

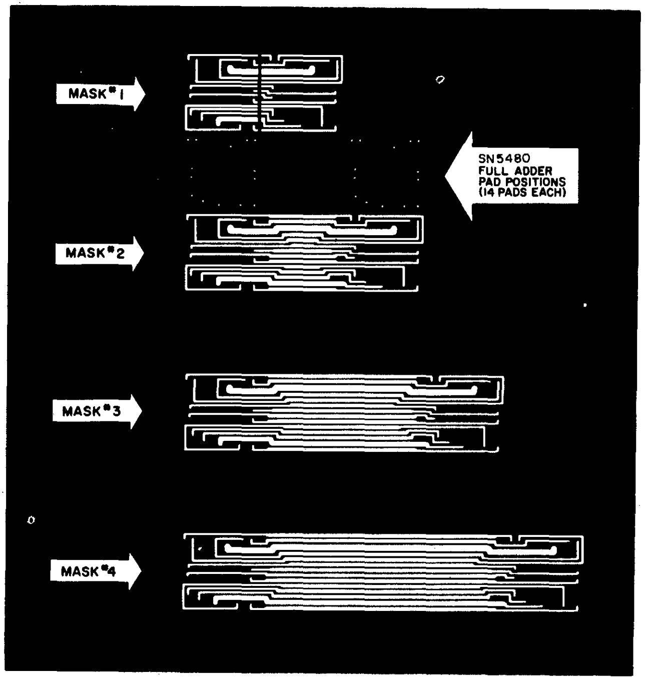
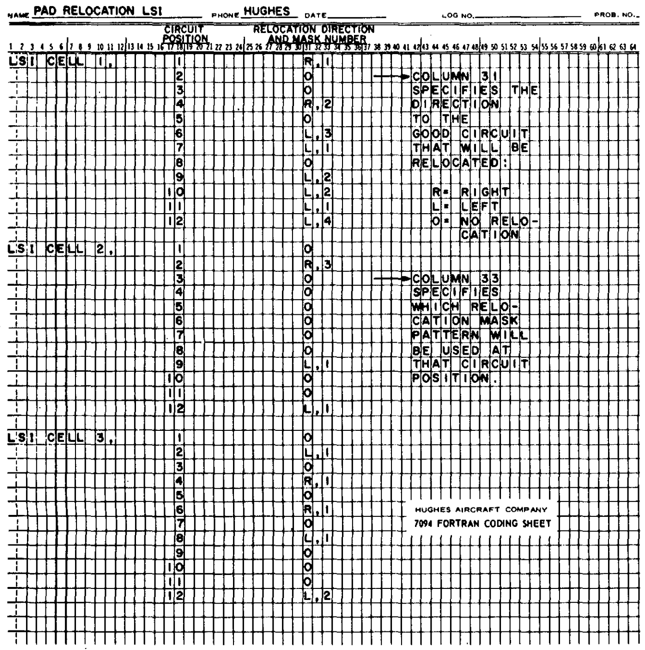


Figure 11—Four relocation patterns for SN5480's

for the three multipliers of Figure 9. Figure 11 shows the four possible Pad Relocation interconnect patterns which are necessary for the LSI multipliers. For these modules it seems appropriate to incorporate simple signal cross-under lines and power distribution in the Pad Relocation level so as to require only two additional levels of interconnect above the tested LSI chips.


NAME PAD RELOCATION LSI PHONE HUGHES DATE LOG NO. PROB. NO.

```
                CIRCUIT       RELOCATION DIRECTION
                POSITION      AND MASK NUMBER
LSI CELL 1,      1            R,1
                 2            O    ---->COLUMN 31
                 3            O         SPECIFIES THE
                 4            R,2       DIRECTION
                 5            O         TO THE
                 6            L,3       GOOD CIRCUIT
                 7            L,1       THAT WILL BE
                 8            O         RELOCATED:
                 9            L,2
                10            L,2           R= RIGHT
                11            L,1           L= LEFT
                12            L,4           O= NO RELO-
                                               CATION
LSI CELL 2,      1            O
                 2            R,3
                 3            O    ---->COLUMN 33
                 4            O         SPECIFIES
                 5            O         WHICH RELO-
                 6            O         CATION MASK
                 7            O         PATTERN WILL
                 8            O         BE USED AT
                 9            L,1       THAT CIRCUIT
                10            O         POSITION.
                11            O
                12            L,1

LSI CELL 3,      1            O
                 2            L,1
                 3            O
                 4            R,1
                 5            O
                 6            R,1       HUGHES AIRCRAFT COMPANY
                 7            O         7094 FORTRAN CODING SHEET
                 8            L,1
                 9            O
                10            O
                11            O
                12            L,2
```

Figure 10—Coding sheet specification

## *A Pad Relocation LSI hardware program*

An LSI hardware development program began in January 1969 (in which Hughes Aircraft Company contracted Texas Instruments to do the multi-level processing) and which resulted in fully tested and packaged 207 gate arrays in May 1969. During this program, (1) TI fabricated and tested one type of their LSI wafers having a certain mix of gates and flip-flops, (2) TI supplied the yield information on each wafer to be processed for Hughes, (3) Hughes generated both the one standard signal interconnect mask for all wafers as well as an individual Pad Relocation mask for each wafer, and (4) using the mask specifications from Hughes, TI processed the two additional levels of interconnect and tested and packaged each of the finished units. Similar programs for higher complexity arrays have since been initiated. The results of this program are described below.

## The logic array to be built in LSI

Investigations were made three years ago at Hughes Aircraft Company into the applicat'on of LSI arrays to techniques for doing the very high speed sum-of-products computations required in advanced digital filtering systems. A result of this study was the development of the high speed "Modular Carry Advance Multiplier" which was described in a 1968 Fall Joint Computer Conference paper by D. F. Calhoun. Among its characteristics is its modularity which allows longer wordlength multiplications to be efficiently accomplished (in terms of speed and parts) simply by paralleling more of the identical modules. A 5-bit sign-and-magnitude Modular Multiplier designed with four types of logic gates and a JK flip-flop was thus chosen as the vehicle for LSI development on this program. Such an array forms and stores in a register the 9-bit sign-and-magnitude product of two 5-bit operands. The 5-bit multiplier design uses 153 NAND gates and 9 flip-flops (each equivalent to six NAND gates) for a total of 207 interconnected gates per LSI wafer.

The logical interconnection of 207 gates using less than one square inch of an LSI wafer represents well any state-of-the-art bipolar LSI approach. Two levels of interconnect (including the Pad Relocation) were used above the tested wafer which already had a first level of metalization for component interconnect. In terms of cross-over complexity, signal linelengths, and circuit fan-outs, the Modular Multiplier design can be considered typical of a 200 gate logic array.

## Description of the chosen LSI slice

The chosen semiconductor slice for this LSI development program was the Texas Instruments type "K" slice. Basically, the K slice is a biploar array of transistor-transistor logic (TTL) gates and flip-flops occupying an active area of about 1.1 square inches. A picture of this LSI wafer is shown in Figure 12. The array is subdivided into 298 cells of dimension 0.084 inch by 0.044 inch. Of the 298 basic wafer cells, 170 are split into two 42 by 44 mil half-cells for gates while the 128 JK flip-flops on the wafer occupy full 84 by 44 mil cells. The distribution of logic elements on the K slice is shown in Figure 13. Each cell labeled "3" has two independent three-input NAND gates while the adjacent cells labeled "5" have an independent five-input NAND gate and a one-input NAND gate. In three of the rows of gates a single seven-input NAND gate designated by a "7" was processed instead of two three-input NAND gates. The rows of full-sized 84 by 44 mil cells contain the JK flip-flops, which are labeled "FF". In total there are 642 logic gates (170 ones, 264 threes, 170 fives, and 38 sevens) and 128 JK flip-flops processed on the wafer.

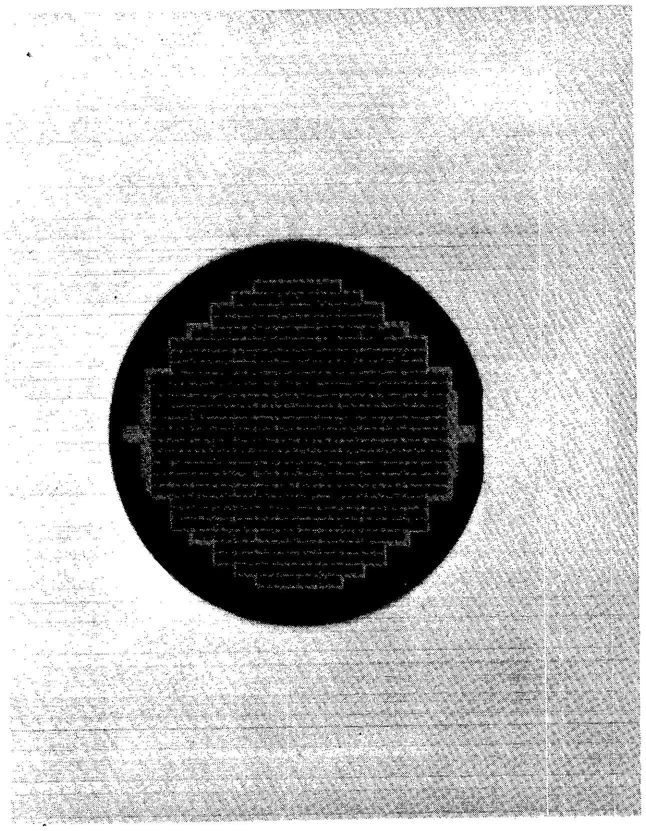


Figure 12—Texas Instruments LSI type "K" slice (HAC Photo 4R07185)

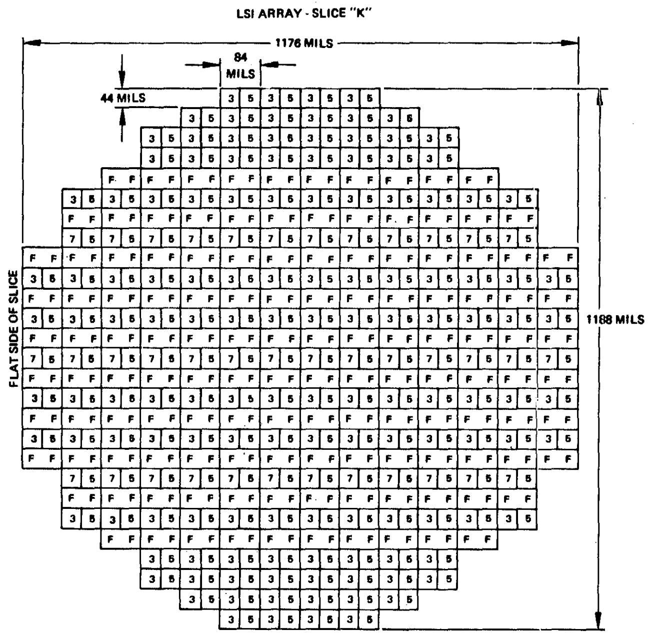


Figure 13—LSI array slice "K"

## Selection of the master pattern and pad relocation patterns

First, a master pattern of circuits was chosen to define the standard circuit positions on the K slice that would be interconnected to form the Modular Multiplier function. This master pattern (shown in Figure 14) was defined with respect to (1) maximizing the probability of successful fulfillment, $P_r(M)$, of the master pattern, (2) facilitating the standard signal interconnect, and (3) using a minimum number of relocation patterns efficiently. After the master pattern and the repertoire of relocation patterns to be used were determined, restricted areas in the Pad Relocation level were defined to allow signal cross-unders from the standard top level signal interconnect. Sufficient cross-under capability for this design was found in the flip-flop cells alone by using certain areas of these cells which are not required by any of the defined relocation patterns. Other cross-under areas can be defined for any more complex designs so as to still use only two metalization layers above the tested circuits. A set of Pad Relocation patterns was prepared to allow the efficient selection of the particular patterns and their positions necessary to fulfill each wafer's master pattern. The chosen set of K slice relocation patterns is shown in Figure 15. This semiautomated specification has facilitated a very fast turnaround and low cost capability for the generation of Pad Relocation masks and for working with new routing requirements, wafer layouts, and logic designs.

## LSI program results

The end results of the Hughes effort described in this section were the two metalization mask specifications used by TI to process each wafer. Only one of these is unique since the use of Pad Relocation allows all signal interconnect to be obtained from a once-generated standard mask. Figure 14 shows the worksheet specification of how the yield of a typical LSI slice can be tailored to the chosen master pattern. The lines with arrowheads at the end specify relocation patterns from the set of patterns shown in Figure 15. The completion of the K slice master pattern was accomplished successfully on each of the 30 wafers attempted. A typical time for a man to complete and verify the specification shown in Figure 14 was two minutes manually.

From the specifications like those in Figure 14, the necessary relocation patterns were selected from the standard set shown in Figure 15 and were added to

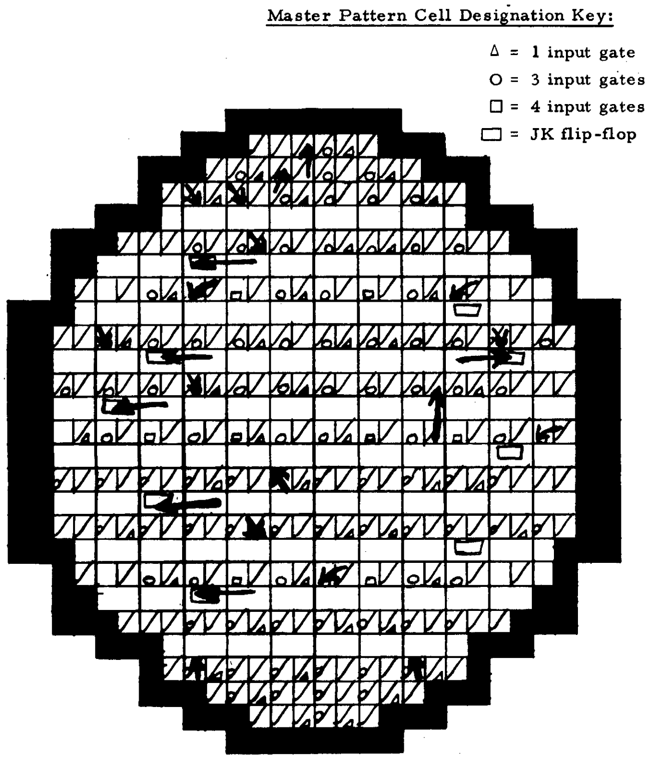


Figure 14—Pad Relocation worksheet with master pattern locations shown

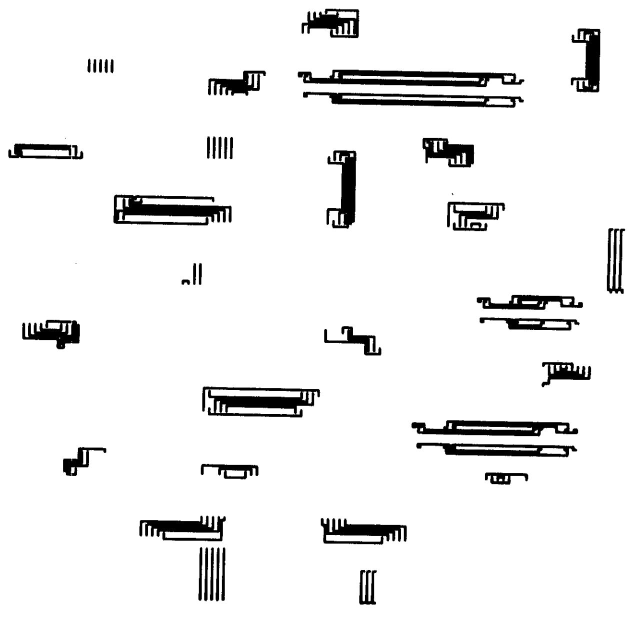


Figure 15—Set of K slice relocation patterns

the standard cross-under pattern to complete the Pad Relocation mask such as the one shown in Figure 16. Only the particular circuit relocation patterns vary within this mask which allows the least possible variation of interconnect and testing from one array to another. The more complex but standard mask is the one shown in Figure 17 which accomplishes all necessary signal interconnect (except the cross-unders to the Pad Relocation level) and the power distribution for the 5-bit multiplier design. The design for this mask can efficiently be done manually for arrays of this and larger size since the master pattern is well distributed. In mask plotting time alone, the Pad Relocation mask required only about 20 percent the time required to plot the signal interconnect metalization patterns. A photograph of the final 207 gate LSI multiplier is shown in Figure 18.

*Statistics of Pad Relocation master patterns*

The choice of a master pattern for Pad Relocation is important since its definition affects the average number of relocated circuits (and thus the routing time and mask complexity) as well as the number and simplicity of the signal interconnect levels. Also a good statistical match between the master pattern and the expected wafer yield distribution will result in a higher probability of successful relocation. As an example, consider a master pattern that is defined too densely about a wafer's periphery. Since peripheral wafer circuits show a much lower yield than the more central

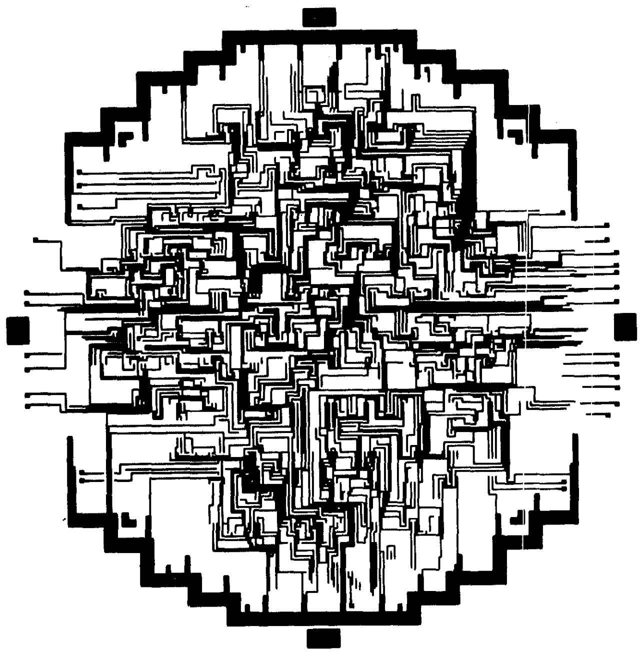


Figure 17—5-bit modular multiplier standard interconnect mask

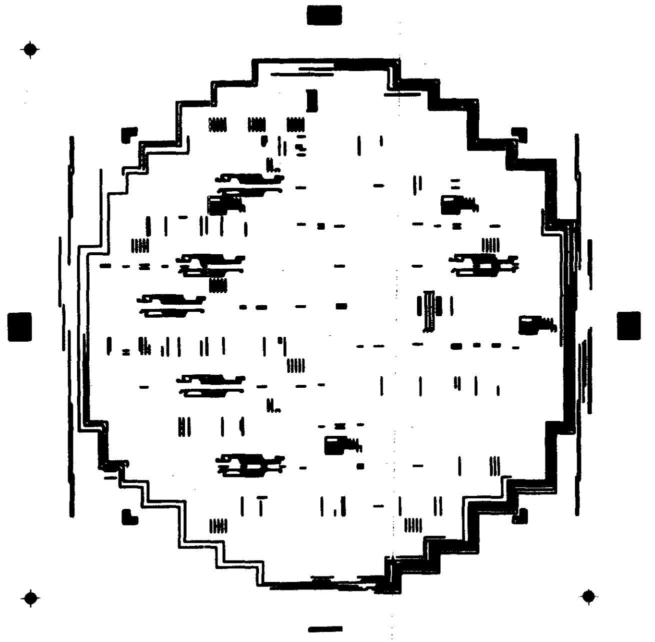


Figure 16—Pad relocation mask with standard cross-unders

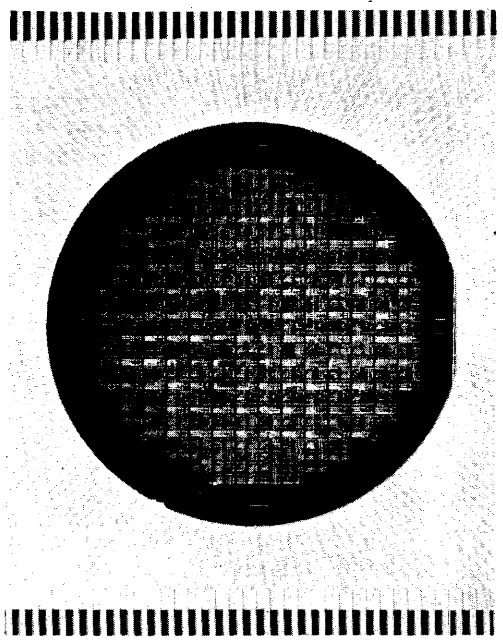


Figure 18—207 gate multiplier LSI array using Pad Relocation (HAC Photo 4R09152)

ones, there will statistically be more relocations, longer relocation lengths, more difficulty in satisfying the master pattern, and a higher concentration of signal interconnect above the master pattern than if the master pattern had been chosen to match the "expected" yield distribution as was done for the example shown in Figure 3.

A first question that must be answered is what is the "expected" yield distribution? Investigations thus far have pointed out only that there is a significant decrease in yield as a function of the distance from the wafer center which can be attributed to boundary defects, and that when good or bad circuits occur, there is a more than random clustering effect. No ability to predict the locations of these clusters has been obtained. What must be done is to examine the yield of large samples of the wafer types that will be used to determine the distribution that best describes their expected yield patterns. This distribution will be different for different ranges of yield as well as for different circuit complexities and wafer types. The master pattern for a specific range of yield, wafer type, and wafer size should be matched to the expected distribution so as to take advantage of any knowledge of where good circuits are more probable. By so doing, the probability of successfully fulfilling a master pattern is maximized while minimizing the expected length of the longest relocations.

Statistical techniques have been developed to determine and compare the efficiency of various master patterns in terms of maximizing both the utilization of good circuits and the probability of successfully fulfilling the master pattern. For example, if y is the percentage of the total circuits that were found to be good (i.e., the yield), m the percentage of total circuits that are in the master pattern, and r the number of unused circuits from which a relocation could be made to each master pattern circuit, then the probability of successfully fulfilling each master pattern circuit independently is:

$$\mathrm{P}(1) = \mathrm{y} + (1-\mathrm{y})\mathrm{y} + (1-\mathrm{y})^2\mathrm{y} + \cdots + (1-\mathrm{y})^r\mathrm{y} = \mathrm{y}\sum_{k=0}^{k=r}(1-\mathrm{y})^k \qquad (1)$$

where the first term is the probability that the master pattern circuit itself is good, and each succeeding term is the conditional probability of needing to examine another candidate for relocation times its probability of being good. Equation (1) can be simplified as follows:

$$\mathrm{y}\sum_{k=0}^{k=r}(1-\mathrm{y})^k = \mathrm{y}\sum_{k=0}^{k=r}\frac{(1-\mathrm{y}-1)(1-\mathrm{y})^k}{(1-\mathrm{y}-1)}$$

$$= \mathrm{y}\sum_{k=0}^{k=r}\frac{(\mathrm{u}-1)\mathrm{u}^k}{-\mathrm{y}} \qquad (2)$$

with

$$\mathrm{u} = (1-\mathrm{y})$$

and

$$\mathrm{y}\sum_{k=0}^{k=r}\frac{(\mathrm{u}-1)\mathrm{u}^k}{-\mathrm{y}} = -\sum_{k=0}^{k=r}(\mathrm{u}-1)\mathrm{u}^k$$

$$= -(\mathrm{u}^{r+1}-1) = 1-(1-\mathrm{y})^{r+1} \qquad (3)$$

therefore,

$$\mathrm{P}(1) = 1-(1-\mathrm{y})^{r+1} \qquad (4)$$

If the master pattern has a total of $M$ circuits in it, then the joint probability of successfully fulfilling all of the $M$ circuits becomes:

$$\mathrm{P(M)} = \mathrm{P}(1)^M = [1-(1-\mathrm{y})]^{r+1M} \qquad (5)$$

Equation (5) is based on an uncorrelated and pseudo-random distribution of good circuits (see Reference 10 with $\mathrm{Y} \geq 0.25$) as well as the same assumption as Equation (1) that there are $r$ circuits (good or bad) for each master pattern circuit fsom which a relocation can be made independently of the other master pattern circuits. It is, however, an unnecessary restriction to assign $r$ circuit positions which could only be used to fulfill each master pattern circuit. Instead, consider successively examining up to $r$ circuit positions which are the closest to each particular master pattern position and, for which, there is still a free path in the Pad. Relocation level to the master pattern position. Then Equation (5) will give the probability of successfully relocating (if necessary) to each of the $M$ required master pattern positions at least one of the $r$ closest and free circuit positions.

Equation (5) determines a family of curves for $P_r(M)$ versus $M$ for various yields and values of $r$. Figure 19 shows the curves of $P_r(M)$ versus $M$ with $y = 0.5$ for $r = 4$ and $r = 9$. It should be noted that each circuit of $M$ may actually be many interconnected gates of logic and $M = 100$ would represent 1000 gates

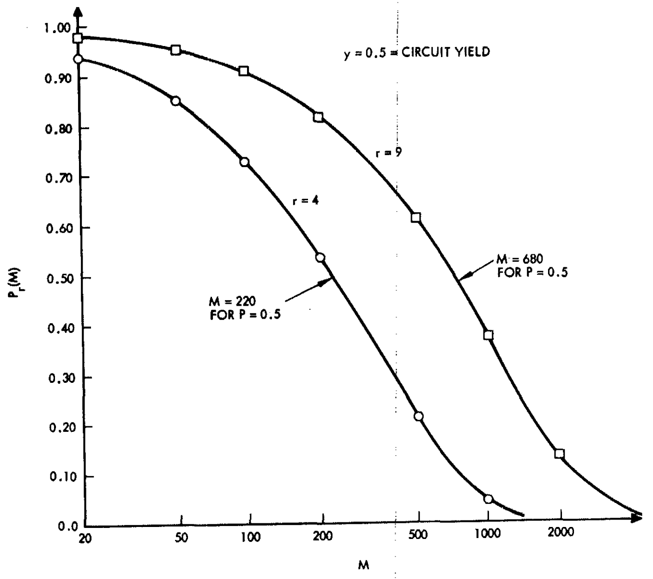


Figure 19—The probabilty $P_r(M)$ of successfully fulfilling a master pattern of M circuits by relocating from one of up to r nearby circuits. Each circuit is a tested unit which may have many gates of logic complexity

if each circuit of $M$ had 10 gates of equivalent logic complexity. If it is desired to successfully fulfill the master patterns of at least half the wafers considered, Figure 19 shows that 220 circuits (and thus probably 750 or more gates) can be used if $r = 4$, and 680 circuits can be used if $r = 9$. Of course, any wafers for which the master pattern was not easily fulfilled are not lost since they can be inventoried and used for other master patterns, or for integrated circuits, or diced and bonded to substrates. As a comparison the most complex current bipolar discretionary unit has an equivalent $M$ of 169 while the 100 percent yield approach has reached an equivalent M of only 24.

### *Advantage of Pad Relocation to LSI signal interconnect*

The prime advantage of Pad Relocation LSI which has been described above is that it places the pads of all used circuits in standard positions which both allows fixed-pattern signal routing between these circuits as well as the utilization of more circuits than allowed by other LSI techniques. There are further advantages, however, to the routing of the standard signal interconnect. For example, the positions to which circuit pads will always be brought can be modified and optimized to facilitate the necessary routing of signals as well as to minimize the lengths of the longest or the most critical signal paths. This will also allow the standard signal interconnect to be designed to require the minimum number of levels and the minimum area per level. Thus, chip areas can be less interconnect limited.

### *Improvement of testing and reliability of large scale integrated systems*

Semiconductor device reliability, as well as propagation delay, is highly dependent on proper maintenance of junction temperatures within certain bounds. From the maximum specified junction temperature, a maximum power dissipation per wafer area can be computed which is dependent on the heat conductive characteristics of the wafer and the cooling techniques used, as well as on the area and power dissipation of the particular circuits. Thus there will be a maximum number of circuits that should be powered up on the wafer. In addition, no region of the wafer should exceed a certain maximum power density in order to insure that the wafer will not have relative "hot spots" where too many powered circuits are located. Pad Relocation LSI can help insure that the wafer power dissipation density is not excessive by specifying the relocated circuits to be primarily those from areas of sparce circuit utilization, thus obtaining a more uniform power density across the entire wafer. By so doing, the system cooling requirements can be relaxed and/or more circuits can be used on the same wafer. This more uniform power dissipation could be quite difficult to insure with other routing techniques since there is less choice in the used circuit positioning. A simple means by which a Pad Relocation computer program could insure a uniform power density would be to either count the number of powered circuits in various wafer regions as the Pad Relocations were being assigned or to assign all Pad Relocations, compute local power densities, and then reassign any necessary Pad Relocations to meet the maximum local power density.

A most significant advantage to Pad Relocation LSI is that test pads can very easily be placed in standard positions in the top layer of wafer metalization. Since they are in standard positions, these test pads can readily be used to facilitate automated probe testing of interconnected wafers just prior to final encapsulation without requiring a large number of additional package leads. Especially for sequential arrays this will be important since it will both allow the pre-setting of the flip-flops to known states and the monitoring of their outputs without adding package leads. It is well known in testing theory that only by having control over the states of flip-flops can it be guaranteed that combinatorial-like tests will be found for a logic array, if they exist. Thus, the ability to define standard test pads will allow both automated probe testing as well as making the definition and execution of the required test sequences simpler.

The definition of standard test pads has further applicability to systems partitioning, improvement of effective processing yield, fault diagnosis, and the testing of redundant networks.

## ACKNOWLEDGMENT

The author sincerely wishes to acknowledge the helpful counsel and encouragement of Professor Lawrence P. McNamee of UCLA, and of Dr. Ira Terris and Messrs. R. F. Stewart, J. M. Block, M. May, and J. S. Steiner, all of Hughes Aircraft Company. The work presented in this paper is part of the author's doctoral research at UCLA supported by Hughes Aircraft Company and conducted under Professor McNamee.

## REFERENCES

1 H R BEELITZ H S MULLER R J LINHARDT R D SIDMAN
*Partioning for large scale integration*
International Solid-State Circuits Conference Digest 1967 Feb 1967 50-57

2 J O CAMPEAU
*The block oriented computer*
Computer Group Conference Digest IEEE New York June 1968 57-60

3 D F CALHOUN
*High speed modular multiplier and digital filter for LSI development*
Proc FJCC Vol 31 847-855

4 H G CRAGON et al
*Large scale integrated circuit arrays*
AF33(615)3546 Texas Instruments Dallas Texas 1966-1969

5 A G F DINGWALL
*High-yield processing for fixed interconnect large scale integrated arrays*
IEEE Transactions on Electron Devices Vol ED-15 Sept 1968 631-637

6 L HAZLETT
*The polycell approach to large scale integration*
Electronics February 20 1967

7 G B HERZOG et al
*Large scale integrated arrays*
AF33(615)3491 Radio Corp America Princeton N J 1966-1968

8 J W LATHROP R S CLARK J E HULL R M JENNINGS
*A discretionary wiring system as the interface betweeen design automation and semiconductor array manufacturer*
Proc IEEE Vol 55 Nov 1967

9 R C MINNICK
*Application of cellular logic to the design of monolithic digital systems*
Microelectronics and Large Systems Spartan Books Washington D C 1965 225-247

10 B T MURPHY
*Cost-size optima of monolithic integrated circuits*
Proc IEEE Vol 52 1537-1545 Dec 1964

11 J J PARISER
*Connection considerations with a view toward batch fabrication*
Proc IEEE Batch Fabrication April 1965 263-319

12 R L PETRITZ
*Current status of large scale integration technology*
IEEE Journal Solid State Circuits Vol SC-2 No 4 Dec 1967
Also Proc FJCC Vol 31 1968 65-85

13 W T RHOADES
*System considerations in large scale integration designs*
Nat Electronic Packaging Production Conference 1968 708-719

14 R B SEEDS
*Yields, eonomics and logistic models for complex digital arrays*
Conference Record 1967 IEEE Internat Convention and Exhibition March 20-23 1967

15 L M SPANDORFER
*Large scale integration—an appraisal*
Advances in Computers 1968 179-238

16 E TAMMARU J B ANGELL
*Redundancy for LSI yield enhancement*
IEEE Journal Solid State Circuits Vol SC-2 No 4 Dec 1967

17 R E THUN R L DONNERSTEIN
*A hybrid circuit approach to LSI*
Digest of Government Microcircuit Applications Conf Wash., D. C.
Vol 1 Oct 1968 169-171

# A consideration of the application of cryptographic techniques to data processing

*by* R. O. SKATRUD

*IBM Corporation*
Research Triangle Park, North Carolina

## INTRODUCTION

Two digital cryptographic techniques are described which may have potential applications in Data Processing Systems. A method of digital substitution analogous to a Vernan double tape system is presented, using a controlled combination of data and the contents of two memories. The second method uses a digital route transposition matrix using a combination of row and column transposition under memory control. Possible ways of achieving key leverage in each ciphering process are described.

The large growth in digital computers and computer usage proliferating to time-shared remote systems presents an increasing need to provide data security within a system as well as applying it to data transmitted over communications media.[1] Two fundamental approaches to producing security in data use are developed in this presentation. One is a digital-substitution technique and the second involves a digital-matrix transposition.

Some of the earliest practical cryptographic systems were the monoalphabetic substitution systems used by the Romans.[2] In these, one letter is substituted for another. For example, an A might be replaced by a C. By the fifteenth century, an Italian by the name of Alberti came up with a technique of cryptoanalyzing letters by frequency analyses. As a result, he invented probably the first polyalphabetic substitution system using a cipher disk. Thus, he would encode several words with one substitution alphabet, then he would rotate the disk and encode several more words with the next substitution alphabet.

Early in the sixteenth century Trithemius, a Benedictine Monk, had the first printed book published on cryptology. Trithemius described the square table or tableau which was the first known instance of a progressive key applied to polyalphabetic substitution. It provided a means of changing alphabets with each character. Later in the sixteenth century, Vigenere perfected the autokey: a progressive key in which the last decoded character led you to the next substitution alphabet in a polyalphabetic key. These were basically the techniques that were widely applied in the cryptomachines in the first half of the twentieth century. Various transposition techniques have been employed including the wide use of changing word order and techniques such as rail transcriptions (used in the Civil War).

In 1883, Auguste Kerckhoffs, a man born in Holland but a naturalized Frenchman, published a book entitled *La Cryptographic Militaire*. In it, he established two general principles for cryptographic systems. They were:

1. A key must withstand the operational strains of heavy traffic. It must be assumed that the enemy has the general system. Therefore, the security of the system must rest with the key.
2. Only cryptoanalysts can know the security of the key. In this, he infers that anyone who pro-

poses a cryptographic technique should be familiar with the techniques that could be used to break it.

From these two general principles, six specific requirements emerged in his book:

1. The key should be, if not theoretically unbreakable, at least unbreakable in practice.
2. Compromise of the hardware system or coding technique should not result in compromising the security of communications that the system carries.
3. The key should be remembered without notes and should be easily changeable.
4. The cryptograms must be transmittable by telegraph. Today this would be expanded to include both digital intelligence and voice (if voice scramblers are employed) utilizing either wire or radio as the medium.
5. The apparatus or documents should be portable and operable by a single person. This requirement is met in the systems proposed in this paper by the portability of the key in a dense storage medium (such as magnetic tape), installable in a processing system by one man.
6. The system should be easy, neither requiring knowledge of a long list of rules nor involving mental strain. In the proposed systems, the key is an automatic-machine-controlled process until a key change occurs.

In 1917 Gilbert S. Vernan, a young engineer at American Telephone and Telegraph Company, using the Baudot code (teletype) invented a means of adding two characters (exclusive or). Vernan's machine mixed a key with text as illustrated by the following:

| | | | | | |
|---|---|---|---|---|---|
| Clear Text | 1 | 0 | 1 | 1 | 1 |
| Key | 0 | 1 | 0 | 1 | 0 |
| Coded Character | 1 | 1 | 1 | 0 | 1 |

To derive the text from the coded character, all that was required was the addition of the key again to the coded character.

| | | | | | |
|---|---|---|---|---|---|
| Coded Character | 1 | 1 | 1 | 0 | 1 |
| Key | 0 | 1 | 0 | 1 | 0 |
| Clear Text | 1 | 0 | 1 | 1 | 1 |

His machines used a key tape loop about eight feet long which caused the key to repeat itself over a high volume of traffic. This allowed cryptoanalysts to derive the key. William F. Friedman, in fact, solved cryptograms using single-loop code tapes but appears to have been unsuccessful when two code tapes were used. Major Joseph O. Mauborgne (U. S. Army) then introduced the one-time code tape derived from a random noise source. This was one of the first theoretically (and in practice) unbreakable code systems. The major disadvantage of the system was the enormous amounts of key required for high-volume traffic.

During the 1920's and 1930's, the rotor-code machines having five and more rotors, each rotor representing a scrambling step, were developed. They proved relatively insecure, requiring only high-traffic volume for the cryptoanalyst to break them. In fact, the Japanese used a code-wheel-type machine for their diplomatic communications well into World War II. It was vulnerable to cryptoanalysis, and William F. Friedman and his group not only solved the code but reconstructed a model of the machine to break Japanese diplomatic correspondence. Thus, President Roosevelt and others were aware of the impending break in diplomatic relations with Japan just prior to World War II.

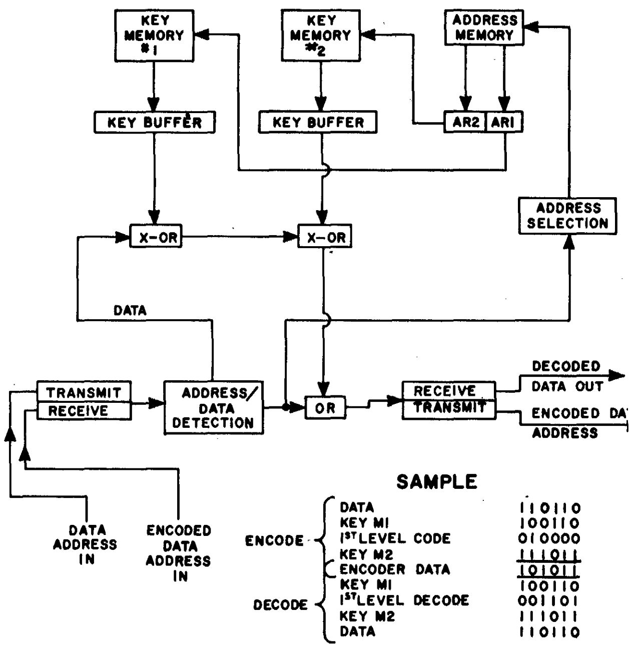
The code wheels (or rotors) were nothing more than key memories storing quantities of key which could easily be changed by interchanging rotor positions, specifying various start points for each rotor, and periodically replacing a set of rotors. This provided a means of producing what I will call key leverage.

### *Digital substitution*

A system which uses the "exclusive or" technique developed by Gilbert S. Vernan, applied directly to data stored and distributed by a computer, is shown in Figure 1.

Instead of using two tapes, this system would use two key memories and an address memory. Synchronization would be achieved by use of the address memory which would be addressed by the first transmitted intelligence. The contents of the two addresses obtained could come into the address registers which would pull key words from the associated addresses in each of the two key memories. Data to be transmitted would be first exclusive ORed with the contents of the first memory location and then with the contents of an address of the second memory. Each character transmitted would thus be encoded twice.

This would represent an element of security dependent on the contents of the two key memories. Order-of-address usage of the key would be dependent on the contents of the address memory. To derive the key, contents of the key memories and address memory would have to be solved. The larger the memory



SAMPLE

| | | |
|---|---|---|
| ENCODE | DATA | 110110 |
| | KEY M1 | 100110 |
| | 1ST LEVEL CODE | 010000 |
| | KEY M2 | 111011 |
| | ENCODER DATA | 101011 |
| DECODE | KEY M1 | 100110 |
| | 1ST LEVEL DECODE | 001101 |
| | KEY M2 | 111011 |
| | DATA | 110110 |

Figure 1—Digital substitution logic

contents, the harder these would be to determine. A large volume of traffic, where starting points in the address control memory would be repeated, could begin to provide clues that could be used to derive the key. Therefore, one would, at frequent intervals determined by usage, change the content of the address memory.

At less frequent intervals, one would change the contents of the key memories. These intervals would be chosen again on the basis of data traffic using the system and the type of security expected from the system.

The relative security of the system would be a function of the amount of memory. If a memory of $n$ bits is considered, total permutations available in the memory of those bits would be $2^n$. If the key is derived from a random noise source, probabilities of getting all o's or all l's in the memory are very small, as would large imbalances existing between o's and l's. Therefore, each key memory would have a distribution in the total bit field available approximating a distribution of bits whose permutations in practice would be more in the neighborhood of $2^{n/2}$. Each of the two key memories would have one of that many practically usable permutations, each one of which could operate on the other in the encoding process. Therefore, by probability theory, the probable permutations would be the product of the two memorypo tentials or $(2^{n/2})$ $(2^{n/2})$ or a potential key field of $2^n$ permutations.

The $2^n$ possible permutations of the key memories would also be acted on by the m addresses of each memory which would all exist in any order in the address memory. Possible permutations of addresses, taking them m at a time, would be m factorial for each of the memories. Therefore, one would achieve the possibility of each of the m-factorial, possible addresses for one memory being able to operate on each of the m-factorial, possible permutations of the other memory. This would represent a total of $(m!)^2$ possible permutations of the addresses.[3]

Therefore, if one were to completely break the key, one would have to derive the one permutation used out of a potential of a total possible equal to $(m!)^2 2^n$. Heavy traffic on the system, with repetition of the key, would however, give handles to the cryptoanalyst in deriving the key so it could not be considered unbreakable.

It is possible to achieve a system which would be unbreakable in theory and achievable without using great amounts of key. This is achievable by using a one-time key with techniques of producing key leverage. Since two memories are used for key, and each memory has addresses associated with it in the address memory, one can achieve key leverage by the fact that different combinations of the contents of the addresses of the key give different coding results. Proper choice of address usage in the address memory will insure that each message that is transmitted would be encoded with a unique code until all the combinations of the addresses were used for the two key memories.

It is known that the Address Memory contains one of m! permutations possible in the m addresses for each key memory. If it is assumed that each memory location contains a character in key memory, that somewhere in the address memory is the address of that character, and that each address is one address memory location, then a practical means of control begins to emerge. If, for example, m is considered to be 1,000 addresses and a usage scheme is used similar to that outlined in Table I, synchronization would be achieved by message numbering consecutively from 000 to 999.

The first character transmitted would use the contents of address 000 for Key Memory 1 and 000 for Key Memory 2. The second character transmitted would use the contents of address 001 in Address Memory 1, and the contents of address 001 in Address Memory 2. This progression could continue to address-memory-location 999 for each of the two address-memory slots.

The second message transmitted would be numbered 001. The address-register pairs for message number 2 would now be 000, and 001 for the first character. The

TABLE I—Address memory usage

| *Message Number* | *Address Memory 1* | *Address Memory 2* |
|---|---|---|
| 000 | 000 | 000 |
| | 001 | 001 |
| | 002 | 002 |
| | ∫ | ∫ |
| | 999 | 999 |
| 001 | 000 | 001 |
| | 001 | 002 |
| | 002 | 003 |
| | ∫ | ∫ |
| | 999 | 000 |
| 002 | 000 | 002 |
| | 001 | 003 |
| | ∫ | ∫ |
| | 999 | 001 |
| ∫ | ∫ | ∫ |
| 500 | 000 | 500 |
| | 001 | 501 |
| | ∫ | ∫ |
| | 999 | 499 |
| 999 | 000 | 999 |
| | 001 | 000 |
| | 002 | 001 |
| | ∫ | ∫ |
| | 999 | 998 |

Refresh Address Memory and repeat cycle.

second character would be 001 and 002. The address register would therefore be using different address pairs for the second message than it did on the first.

At the 501st message, it would bear number 500. The address pairs for the first character transmitted would now be 000 and 500. The second character transmitted would use address-memory locations 001 and 501.

Therefore, it can be seen that by continuing the sequence through message 1000 bearing number 999, no repetition of address pairs will exist. Therefore, with m equal to 1,000 and two key memories and 2 address memories, the system limit—if used in this way—is 1,000 messages of 1,000 characters each.

At the time the system limit is reached, one would change the address memory by supplying a new permutation of addresses for each of the two address-memory slots. This would provide the capability of transmitting and receiving another 1,000 messages of 1,000 characters each. It can be seen that the system employs a progressive-key system and, in theory, one could use m! combinations of addresses in each of the two address-memory slots before obvious key repetition would begin, without changing the contents of the key memories. In practice however, one would, at predetermined intervals, change the contents of the key memories.

It can be shown that the system is modular. By adding a third key memory and a third address memory slot, the system would be expanded to 1,000,000 messages each with a capacity of 1,000 characters. It can also be shown that a trade-off exists between message length and number. For example, if message length were defined to be a maximum of 100 characters instead of 1,000 the message count on the expanded system could go to 10,000,000 messages before the key would be repeated.

In a system using two levels of encoding and m = 1,000 at a transmission rate of 2,000 bits per second, the key will last for 1.4 hours of continuous transmission before the address slots in the address memory would have to be changed. This assumes that 10 bits are present in each key memory address. If transmission loading was 50%, this figure would go to 2.8 hours. Therefore, with heavy traffic, the Address Memory Contents would have to be changed two or three times per day. This could be arranged by pre-storing numbers of changes on a dense-storage medium such as magnetic tape.

Higher usage rates would require higher rates of change for the Address Memory and/or a modular expansion of the key system. Therefore, the system is applicable to any rate of key usage that is in use today. It is also modular, as can be seen, by choice of m.

Thus, it is possible to use a system of digital substitution in a cryptographic computer system which would, if system design parameters were properly chosen, deny access to data in a system to all who did not possess the cryptographic key. The system described here is basically a polyalphabetic substitution system. It employs the fundamental techniques employed by Vernan and would also include some of the characteristics of the rotor machines in achieving leverage in the number of permutations available on data. It is, however, different since we are now operating on the digital makeup of the intelligence rather than on the character as an entity, and we use electronics instead of the mechanical rotor. We also avoid repetitive use of the key which was the reason that rotor-machine codes were finally broken.

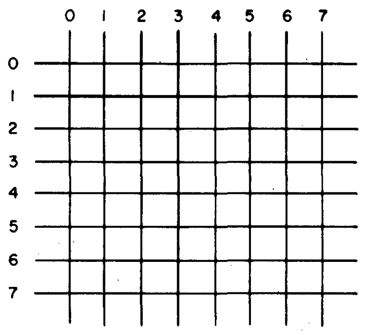
*Digital route transposition*

Transposition techniques can also be used in conjunction with data processing. If the route transposition technique is applied to the read-in and read-out of digital data from a matrix, it is possible to achieve the results of polyalphabetic substitution without a direct substitution key being required. It can be shown that key usage is far less than that required for direct substitution. With the data-key leverage obtained, some interesting possibilities on key transmission can be obtained. With these, it becomes more feasible to explore the possibility of single-use keys.

To illustrate the method, let us consider an $n^2$ matrix where $n = 8$. The matrix will be made up of 8 rows and 8 columns. Information can be read into and out of the 8 columns of the matrix in any order.

The information would be transmitted into the receiving-matrix columns in the same order that it left the transmitting matrix. To complete the data reconstruction, the information in the receiving matrix now would be read out in the same row order that it entered the transmitting matrix. Therefore, the process is reversible.

Figure 2 shows the base matrix. If an 8-by-8 matrix is considered, there are 8 factorial different orders possible in both the rows and columns. For any one matrix of information (64 bits), there are a possible $(8!)^2$ ways of seeing this information when transmitted.[2] Eight factorial squared gives an approximate $1.6 \times 10^9$ possible



**ROW CONTROL**

8! = 40,320
$2^{16}$ = 65,536
∴ 16 BITS REQUIRED

**COLUMN CONTROL**

8! = 40,320
$2^{16}$ = 65,536
∴ 16 BITS REQUIRED

Figure 2—Digital route transposition matrix

TABLE II—Effect of matrix size on permutations

| *Matrix Size: n* | *Read-Write Permutations:* $(n!)^2$ |
|---|---|
| 1 | 1 = $1.0 \times 10^0$ |
| 2 | 4 = $2.0 \times 10^0$ |
| 3 | 36 = $3.6 \times 10$ |
| 4 | 576 = $5.8 \times 10^2$ |
| 5 | 14,400 = $1.4 \times 10^4$ |
| 6 | 518,400 = $5.2 \times 10^5$ |
| 7 | 25,401,600 = $2.5 \times 10^7$ |
| 8 | 1,625,702,400 = $1.6 \times 10^9$ |
| 9 | 131,681,894,400 = $1.3 \times 10^{11}$ |
| 10 | 13,168,189,440,000 = $1.3 \times 10^{13}$ |
| 11 | = $1.6 \times 10^{15}$ |
| 12 | = $2.4 \times 10^{17}$ |
| 13 | = $3.9 \times 10^{19}$ |
| 14 | = $7.7 \times 10^{21}$ |
| 15 | = $1.7 \times 10^{24}$ |
| 16 | = $1.4 \times 10^{26}$ |

permutations on each matrix. Since it would be the function of the key to select each matrix permutation, each matrix would be transmitted with a different key. Table II shows the effect of varying matrix size in terms of available permutations on the data.

The elements of control required for the rows and columns of the matrix must be independent. To keep control independent, Row Key and Column Key Memories can be used. Since in the example chosen there are 8 rows and 8 columns, there are 8! different possible orders to read into or out of each matrix. Thus $2^{16}$ is approximately equal to 8!. It can be shown that 16 bits will be sufficient for reading information into or out of the rows of the matrix. Likewise, 16 bits will allow information to be read into or out of the 8 columns. Therefore, 32 bits of key are required to encode and decode 64 bits of information using a Digital Route Transposition Matrix.

By looking at direct key usage and comparing it to key usage described on Digital Substitution, we find that key consumption per transmitted bit is reduced by a factor of 4. For each 64 bits transmitted by the Digital Route Transposition Matrix, 32 bits of key are used. With Digital Substitution 2 bits of key are used for each bit transmitted.

If 1,000 addresses are assumed in each of two address slots in Address Memory, the potential for applying them to something analogous to message number again exists. However, since each step of the Address Memory now transmits a full matrix of information, synchronization would now be achieved by matrix count instead of message number. Therefore, one would step through the address count. If 1,000 addresses were used in each of the row and column memories, one would step through the address-register counts 1,000,000 times, pairing up a different row and column address count every step. In terms of usable bits available for transmission, this would yield 64,000,000 data bits. If the system were transmitting 2,000 bits per second, this would represent 8.8 hours of continuous transmission. If transmission utilization was 50%, this would represent 17.6 hours of transmission. It would be applicable to any data rate by varying the address memory-change rate. At the end of the 64,000,000 bits, the address memories would be refreshed with a new permutation and the process would continue. Thus, no key repetition would occur.

If data rates were very high, one could consider transmitting the address information encoded in the one-time key. This could be accomplished by the addition of an Address Buffer Memory which would be loaded prior to the point where the system ran out of address permutations. At the point of run-out, a new permutation would be moved out of the Address Buffer into the Address Memory and the process would continue. Since both ends of a system must be synchronized, the transfer would always occur simultaneously at both ends of the system.

After some predetermined number of address-permutation changes, the key would be changed in the Row and Column Key Memories. This change would not have to be frequent unless a key compromise was suspected. Frequency of change will normally be determined by choice of memory size and other design parameters in the system.

Figure 3 represents a diagram of the transmitting function for a Digital Route Transposition Matrix. An address followed by data would come into the system. The starting address for the address memory would be also transmitted to the receiving station. The station would select a pair of key addresses in the address memory which would pull the contents from Row-Key and Column-Key memories to activate the row- and column-scan selection control.

The row-scan selection control would activate read-in to one matrix of data, which would fill all of its rows with data. At that point in time, column-scan selection control would take over and begin transmitting data from the first filled matrix. While the first matrix is transmitting, a new row-key word would be brought out by stepping the address-selection logic to get the next row key from Row-Key Memory.

When the first matrix has finished transmitting, address-selection would supply the second address for the Column-Key Memory. With proper choice of timing

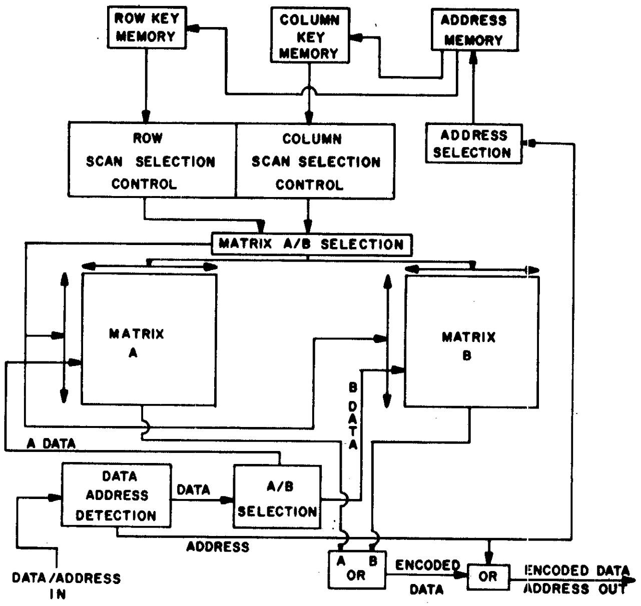


Figure 3—Digital route transposition matrix

relationships, continuous data transmission would occur by permitting encoded data from matrix A or B to enter the line. While one matrix is transmitting, the second would be filling with data.

The receive function would be the reverse of the transmitting function. It can be seen that a double matrix is required to secure continuous transmission.

*Error detection and recovery*

Existing techniques of error detection could be employed. When techniques of ciphering are used on a system employing transmission lines, it would be possible to use a polynomial accumulation which comes very close to being unique for each block of data transmitted. The one- or two-character accumulation would be transmitted at the end of the block and compared to that generated by the received data. An error would be assumed only if there was a difference.

Error recovery would be initiated by transmittal of a negative acknowledgment to that block of data. Action at the transmitting end would then consist of retransmission of the data block with the same address designations for key usage being held. This would be necessary to prevent the same text from being transmitted twice with different keys. Transmission twice with different keys could provide information to a cryptoanalyst which could possibly permit breaking a portion of a key.

Polynomial accumulations would be equally applicable to either the double substitution or transposition technique. Accumulations could be done either before or after encoding, depending on the handling of transmission control characters.

Another possible error detection technique, particularly with a transposition matrix, would be the utilization of horizontal and vertical parity assignments. The reliability of the technique would require evaluation for the particular application to determine what types of multiple-bit errors would result in lack of error detection. It would be used only if the probability of detection were sufficiently high for the application. Again, detection of the error would require re-transmission by the same key-address designators that were used for the original transmission.

Both systems would satisfy all the criteria and principles established by Auguste Kerckhoff's book published in 1883. There would be no requirement for manual intervention unless maintenance was required. Since a one-time code would be used, it is in theory unbreakable. The security of the system rests with the key, not the hardware. In operation, the key is undergoing continuous change automatically and/or under control of an operator, depending on application and specific hardware design. Both systems would be simple, and for all practical purposes, transparent to the operators. Operators would handle the clear information as is done today, even in the most confidential types of operations.

Selection of operators would remain a management function, as it is today. This system would be designed to prevent unauthorized proliferation of confidential information by direct access from other I/O devices that do not possess the key, in spite of the fact that they may have the hardware.

With integrated circuits becoming available and the cost per circuit function decreasing, it becomes possible to consider undertaking designs that would offer relatively high degrees of privacy in computer systems at reasonable cost. High-density memories and the technology to support the logical control of cryptographic systems exist today.

## REFERENCES

1 A WESTIN
*Privacy and freedom*
Atheneum 1967

2 D KAHN
*The code breakers*
The Macmillan Co 1967

3 C H RICHARDSON
*An introduction to statistical analysis*
Harcourt Brace and Co 1934

# Security controls in the ADEPT-50 time-sharing system

*by* C. WEISSMAN

*System Development Corporation*
Santa Monica, California

*"Authority intoxicates/And makes mere sots of magistrates"—Butler*

## FOREWORD

At present, the system described in this paper has not been approved by the Department of Defense for processing classified information. This paper does not represent DOD policy regarding industrial application of time- or resource-sharing of EDP equipment.

## INTRODUCTION

Computer-based, resource sharing systems are, and contain, things of value; therefore, they should be protected. The valuables are the information data bases, the processes that manipulate them, and the physical plant, equipment, and personnel that form the system plexus. An extensive lore is developing on the subject of system protection.[1,2] Petersen and Turn[3] discuss in considerable detail the substance of protection of non-military information systems in terms of threats and countermeasures. Ware[4,5] contrasts "security" and "privacy" for viewing protection in militarys ystems as well. This paper describes the security controls implemented in the ADEPT–50 time-sharing system[6]—a resource sharing system designed to handle sensitive information in classified government and military facilities.*

Our approach to security control is based on a set theoretic model of access rights. This approach appears natural, since the important objects of security are sets of things—users, terminals, programs, files—and the operators of set theory—membership, intersection, union—are easily programmed for, and quickly performed by, computer. The formal model defines time-sharing security control of user, terminal, job and file security objects in terms of equations of access based upon their security profiles—a triplet of Authority, Category, and Franchise property sets. The correspondence of these properties to government and military Classification, Compartments, and Need-to-Know is demonstrated. Implementation of the model in the ADEPT–50 Time-Sharing System is described in detail, as are features that transcend the model including initialization of the security profiles, the LOGIN decision procedure, system integrity checks, security residue control, and security audit trails. Other novel features of ADEPT security control are detailed and include: automatic file classification based upon the cumulative security history of referenced files; the "security umbrella" of the ADEPT job; and once-only passwords. The paper concludes with a recapitulation of the goals of ADEPT security control, approximate costs of implementation and operation of the security controls, and suggested extensions and improvements.

Historically, protection of a sensitive computer facility has been attained by limiting physical access to the computer room and shielding the computer complex

* Development of ADEPT was supported in part by the Advanced Research Projects Agency of the Department of Defense.

from electromagnetic radiation. This "sheltered" approach promotes one-at-a-time, batch usage of the facility. Modern hardware and software technology has moved forward to more powerful and cost/effective time-shared, multi-access, multiprogrammed systems. However, three features of such systems pose a challenge to the sheltered mode of protection: (1) concurrent multiple users with different access rights operating remote from the shielded room; (2) multiple programs with different access rights co-resident in memory; and (3) multiple files of different data sensitivities simultaneously accessible. These features appear to violate traditional methods of accountability based upon a single user (or multiple users with like clearances) operating within strictly controlled facilities. The problem is of such magnitude that no time-sharing system has yet been certified for use in the manner described! However, some multi-access systems are in operation in a classified mode,[7,8] and a number of design approaches have been suggested.[9,10,11,12]

In addition to the usual goal of building an effective time-sharing system,[13] the ADEPT project began with a number of security objectives as well:

1. Build a security control mechanism that supports heterogeneous levels and types of classifications.
2. Design the security control mechanism in such a manner that it is itself unclassified until primed by security configuration parameters, a point strongly supported by Baran[14] regarding communicatons security.
3. Construct the security control mechanism as an isolated portion of the total time-sharing system so that it may be carefully scrutinized for correctness, completeness, and reliability.
4. Do the above in as frugal a manner as possible, considering costs to design, fabricate, and operate. Good system performance is our principal criterion in selecting among alternative technical solutions, as noted by the author elsewhere.[15]

In approaching our task, we recognize security as a total system problem involving hardware, communication, personnel, and software safeguards. However, our focus is primarily on monitor software, and its interfaces with the other areas. This view is not parochial: our hardware is a standard IBM 360 model 50; communication security is an established field of study with considerable technological know-how;[14] and the policy, doctrine, and procedures for personnel behavior in classified environments are extensive, with legal foundations. Thus, our only degree of freedom is the control we build into the time-sharing executive software.

*A security control formalism*

A formal model of software security control for access to sensitive portions of ADEPT is developed here.

### Security objects

Four kinds of security objects are to be managed by our model: user, terminal, job, and file. Let $u$ denote some user; $t$ some terminal; $j$ some job; and $f$ some file.

### Security properties

Each security object is described by a security profile that is an ordered triplet of security properties—Authority (A), Category (C), and Franchise (F). Authority is a set of hierarchically ordered security jurisdictions. Category is a set of discrete security jurisdictions. Franchise is a set of users licensed with privileged security jurisdiction.

The property "Authority" is defined as a set A, where

$$A = \{a^0 < a^1 <, \cdots, < a^\omega\} \tag{1}$$

and the specific members, $a^i$, of the set are security jurisdictions hierarchically ordered.

"Category" is a discrete set of specific compartments, $c^i$,

$$C = \{c^0, c^1, \cdots, c^\psi\} \tag{2}$$

Compartments are mutually exclusive security sanctuaries with discrete jurisdictions.

"Franchise" is a security jurisdiction privileged to a given set of users, i.e.,

$$F = \{u | u \text{ is a user}\} \tag{3}$$

For a given terminal, $t$, let a given Authority set, A, be denoted by $A_t$, or in general, let a given security object, $\alpha$, denote a given property, P, for $\alpha$ as $P_\alpha$. Hence we can speak of $A_u$, or $C_j$, etc., to mean the specific Authority set for a given user, $u$, or the specific Category set for a given job, $j$, respectively.

Four important sets (of users) arise with respect to the Franchise property, namely, Franchise for files, terminals, jobs, and users. To distinguish the sense in which a given user is being considered, we subscript $u$ by the security object under consideration. Hence, $u_f$ means the user with jurisdiction to file $f$; $u_t$ and $u_j$ are similarly defined. For completeness, we define $u_u$ as simply $u$. We can now define Franchise for each security object.

$$F_u = \{u\} \quad (4)$$

$$F_t = \{u_t^0, u_t^1, \cdots, u_t^\lambda\} \quad (5)$$

$$F_j = \{u_j^0, u_j^1, \cdots, u_j^\mu\} \quad (6)$$

$$F_f = \{u_f^0, u_f^1, \cdots, u_f^\nu\} \quad (7)$$

Equation (4) states that the Franchise for a user is restricted to himself; his jurisdiction is unique, and no other user is so endowed. Equation (5) states that the terminal Franchise is possessed by $\lambda$ different users who have jurisdiction over the terminal $t$. Likewise, equations (6) and (7) define the job and file Franchise sets.

In security discussions, one hears the familiar phrase, "he needs a higher-level clearance." We can now define "higher level" with our model.

Let $\alpha$ and $\beta$ be security objects and let $\rho$ be some function such that $\rho(A_\alpha) \epsilon A$.
Then,

$$A_\alpha \geq A_\beta \leftrightarrow \rho(A_\alpha) \geq \rho(A_\beta) \quad (8)$$

$$C_\alpha \geq C_\beta \leftrightarrow C_\alpha \supseteq C_\beta \quad (9)$$

$$F_\alpha \geq F_\beta \leftrightarrow F_\alpha \supseteq F_\beta \quad (10)$$

Equation (8) claims that the Authority of a security object, $A_\alpha$ is at a "higher level" than another security object $A_\beta$ when the specific authority, $a_\alpha$ is greater than the specific authority, $a_\beta$.

It is implicit in equations (1) and (8) that the specific authorities, $a^i$, must be numerically encoded for the magnitude relationships to hold. Equations (9) and (10) define $P_\alpha$ to be greater than $P_\beta$ if and only if $P_\beta$ is a subset of $P_\alpha$.

Events may alter the membership of property sets. Let $P_f^e$ be the $e$th $P_f$ in a given context.

Define the Authority history, $A_h$, at the $e$th event as

$$A_h(0) = a_f^0 \quad (11)$$

$$A_h(e) = \max (A_h(e-1), \rho(A_f^e)), e > 0 \quad (12)$$

Likewise, define the Category history $C_h$, at the $e$th event as

$$C_h(0) = \phi \quad (13)$$

$$C_h(e) = C_h(e-1) \cup C_f^e, e > 0 \quad (14)$$

Equations (11) through (14) recursively define two useful sets that accumulate a history of file references as a function of file reference events, $e$. A history of the highest Authority, $A_h$, is defined by equation (12) as either the previous set, $A_h(e-1)$, or the current set, $\rho(A_f^e)$, whichever is larger in the sense of equation (8). Equation (11) gives the initial condition as some low specific file authority, $a_f^0$. Equation (14) defines the highest Category history as the union of the previous set, $C_h(e-1)$, and the current set, $C_f^e$; while equation (13) states that the union is initially the empty set.

Though $F_h$ could be defined in our model, no need is seen at this time for a Franchise history. More will be said about these history sets later.

## Property determination

Table I presents in a $3 \times 4$ matrix a summary of the rules for determining the security profile triplets, $P_\alpha$. We shall examine these rules here. For the user $u$, $A_u$ and $C_u$ are given constants, and $F_u$ is given by equation (4). For the terminal $t$, $A_t$ and $C_t$ are given constants, and $F_t$ is given by equation (5). Given $A_u$ and $A_t$, we determine $A_j$ as:

$$A_j = \min (A_u, A_t) \quad (15)$$

Likewise, given $C_u$ and $C_t$, we determine $C_j$ as:

$$C_j = C_u \cap C_t \quad (16)$$

Equation (6) gives $F_j$ to complete the job security profile triplet.

An existing file has its security profile predetermined with $A_f$ and $C_f$ as given constants, and $F_f$ as given by equation (7). However, a new file—one just created—derives its security profile from the job's file access history according to the following:

$$A_f = A_h(e) \quad (17)$$

$$C_f = C_h(e) \quad (18)$$

$$F_f = u_j^i \quad (19)$$

From equations (11) through (14) we see how the Authority and Category histories accumulate as a function of event $e$. These events are the specific times when files are accessed by a job. To maintain security

TABLE I—Security property determination matrix

| Object \ Property | Authority A | Category C | Franchise F |
|---|---|---|---|
| User, u | Given Constant | Given Constant | u |
| Terminal, t | Given Constant | Given Constant | $u_t^i$ |
| Job, j | $\min(A_u, A_t)$ | $C_u \cap C_t$ | $u_j^i$ |
| File, f | *Existing file* Given Constant | *Existing file* Given Constant | $u_f^i$ |
| | *New file* $\max(A(_h e-1), \rho(A_f^e)), e > 0$ | *New file* $C_h(e-1) \cup C_f^e, e > 0$ | $u_j^i$ |

integrity, these histories can never exceed (i.e., be greater than) the job security profile. This is specified as,

$$A_h(\infty) \rightarrow A_j \tag{20}$$

$$C_h(\infty) \rightarrow C_j \tag{21}$$

For $e = 0$, we see the properties initialized to their simplest form. However, as $e$ gets large, the histories accumulate, but never exceed the upper limit set by the job. $A_h(e)$ and $C_h(e)$ are important new concepts, discussed in further detail later. We speak of them, affectionately, as the security "high-water mark," with analogy to the bath tub ring that marks the highest water level attained.

The Franchise of a new file is always obtained from the Franchise of the job given by equation (6). When $i = \mu = 0$, the job is controlled by the single user $u_j$ who becomes the owner and creator of the file with the sole Franchise for the file.

## Access control

Our model is now rich enough to express the equations of access control. We wish to control access by a user to the system, to a terminal, and to a file. Access is granted to the system if and only if

$$u \in U \tag{22}$$

where $U$ is the set of all sanctioned users known to the system.
Access is granted to a terminal if and only if

$$u \in F_t \tag{23}$$

If equations (22) and (23) hold, then by definition

$$u = u_t = u_j \tag{24}$$

Access is granted to a file if and only if

$$P_j \geq P_f \tag{25}$$

for properties A and C according to equations (8) and (9), and

$$u_j \in F_f \tag{26}$$

If equations (25) and (26) hold, then access is granted and $A_h(e)$ and $C_h(e)$ are calculated by equations (12) and (14).

## Model interpretation

Three different dimensions for restricting access to sensitive information and information processes are possible with the security profile triplet. The generality of this technique has considerable application to public and military systems. For the system of interest, however, the Authority property corresponds to the Top Secret, Secret, etc., levels of government and military security: Category corresponds to the host of special control compartments used to restrict access by project and area; such as those of the Intelligence and Atomic Energy communities; and the Franchise property corresponds to access sanctioned on the basis of need-to-know. With this interpretation, the popular security terms "classification" and "clearance" can be defined by our model in the same dimensions—as a min/max test on the security profile triplet. Classification is attached to a security object to designate the minimum security profile required for access, whereas clearance grants to a security object the maximum security profile it has permission to exercise. Thus, legal access obtains if the clearance is greater than or equal to the classification, i.e., if equation (25) holds.

Another observation on the model is the "job umbrella" concept implied by equations (22) through (26); i.e., the derived clearance of the job (not the clearance of the user) is used as the security control triplet for file access. The job umbrella spreads a homogeneous clearance to normalize access to a heterogeneous assortment of program and data files. This simplifies the problem of control in a multi-level security system. Also note how the job umbrella's high-water mark (equations (11) through (14)) is used to automatically classify new files (equations (17) and (18)); this subject is discussed further below.

A final observation on the model is its application of need-to-know to terminal access, equation (23). This feature allows terminals to be restricted to special people and/or special groups for greater control of personnel interfaces—i.e., systems programmers, computer operators, etc.

### *Security control implementation*

The selection of a set theoretic model of security control was not fortuitous, but a deliberate choice biased toward computational efficiency and ease of implementation. It permits the clean separation and isolation of security control code from the security control data, which enables ADEPT's security mechanisms to be openly discussed and still remain safe—a point advocated by others.[14,16] We achieve this safety by "arming" the system with security control data only once at start-up time by the SYSLOG procedure discussed later. Also, the model improves the credibility of the security system, enhancing its understanding and thereby promoting its certification.

## Security objects: Identity and structure

Each security object has a unique identification (ID) within the system such that it can be managed individually. The form of the ID depends upon the security-object type; the syntax of each is given below.

### User identification

For generality of definition, each user is uniquely identified by his *user:id*, which must be less than 13 characters with no embedded blanks.

The *user:id* can be any meaningful encoding for the local installation. For example, it can be the individual's Social Security number, his military serial number, his last name (if unique and less than 13 characters), or some local installation man-number convention. The set of all *user:ids* constitutes the universal set, *U*.

### Terminal identification

All peripheral devices in ADEPT are identified uniquely by their IBM 360 device addresses. Besides interactive terminals, this includes disc drives, tape drives, line printer, card reader-punch, drums, and 1052 keyboard. Therefore, *terminal:id* must be a two-digit hexadecimal number corresponding to the unit address of the device.

### Job identification

ADEPT consists of two parts: the Basic Executive (BASEX), which handles the allocation and scheduling of hardware resources, and the Extended Executive (EXEX), which interfaces user programs with BASEX. ADEPT is designed to operate itself and user programs as a set of 4096-byte pages. BASEX is identified as certain pages that are fixed in main core, whereas EXEX and user programs are identified as sets of pages that move dynamically between main and swap memory. A set of user programs are identified as a job, with page sets for each program (the program map) described in the job's environment area, i.e., the job's "state tables." Every job in ADEPT has an environment area that is swapped with the job. It contains dynamic system bookkeeping information pertinent to the job, including the contents of the machine registers (saved when the job is swapped out), internal file and I/O control tables, a map of all the program's pages on drum, *user:id*, and the job security control parameters. The environment page(s) are memory-protected against reading and writing by user programs, as they are really swappable extensions of the monitor's tables.

The *job:id* is then a transitory internal parameter which changes with each user entrance and exit from the system. The *job:id* is a relative core memory address used by the executive as a major index into central system tables. It is mapped into an external two-digit number that is typed to the user in response to a successful LOGIN.

### File identification

ADEPT's file system is quite rich in the variety of file types, file organization, and equipment permitted. There are two file types: temporary and permanent.

Temporary files are transitory "scratch" disc files, which disappear from the system inventory when their parent job exits from the system. They are always placed on resident system volumes, and are private to the program that created them.

Permanent files constitute the majority of files cataloged by the system. Their permanence derives from the fact that they remain inventoried, cataloged, and available even after the job that created or last referenced them is no longer present, and even if they are not being used. Permanent files may be placed by the user on resident system volumes or on demountable private volumes.

There are six file organizations from which a user may select to structure the records of his file: Physical-sequential, S1; non-formatted, S2; index-sequential, S3; partitioned, S4; multiple volume fixed record, S5; and single volume fixed record, S9. Regardless of the organization of the records, ADEPT manages them as a collection, called a file. Thus, security control is at the file level only, unlike more definitive schemes of sub-element control.[8,10–12]

All the control information of a file that describes type, organization, physical storage location, date of creation, and security is distinct from the data records of the file, and is the catalog of the file.

All cataloged ADEPT files are uniquely identified by a four-part name; each part has various options and defaults (system assumptions). This name, the *file:id*, has the following form:

*file:id* : : = *name*, *form*, *user:id*, *volume:id*

*Name* is a user-generated character string of up to eight characters with no embedded blanks. It must be unique on a private volume as well as for Public files (described below).

*Form* is a descriptor of the internal coding of a file. Up to 256 encodings are possible, although only these seven are currently applicable:

| | |
|---|---|
| 1 = | binary data |
| 2 = | relocatable program |
| 3 = | non-relocatable program |
| 4 = | card images |
| 5 = | catalog |
| 6 = | DLO (*De*layed *O*utput) |
| 7 = | line images |

*User:id* corresponds to the owner of the file, i.e., the creator of the file.

*Volume:id* is the unique file storage device (tape, disc, disc pack, etc.) on which the file resides. For various reasons, including reliability, ADEPT file inventories are distributed across the available storage media, rather than centralized on one particular volume. Thus, all files on a given disc volume are inventoried on that volume.

## Security properties: Encoding and structure

Implementation of the security properties in ADEPT is not uniform across the security objects as suggested by our model, particularly the Franchise property. Lack of uniformity, brought about by real-world considerations, is not a liability of the system but a reflection of the simplicity of the model. Extensions to the model are developed here in accordance with that actually implemented in ADEPT.

### Authority

Authority is fixed at four levels ($\omega = 3$ for equation (1)) in ADEPT, specifically, UNCLASSIFIED, CONFIDENTIAL, SECRET, and TOP SECRET in accordance with Department of Defense security regulations. The Authority set is encoded as a logical 4-bit item, where positional order is important. Magnitude tests are used extensively, such that the high-order bits imply high Authority in the sense of equation (8).

### Category

Category is limited to a maximum of 16 compartments ($\psi \leq 15$ for equation (2)), encoded as a logical 16-bit item. Boolean tests are used exclusively on this datum. The definition of (and bit position correspondence to) specific compartments is an installation option at ADEPT start-up time (see SYSLOG). Typical examples of compartments are EYES ONLY, CRYPTO, RESTRICTED, SENSITIVE, etc.

### Franchise

Property Franchise corresponds to the military concept of need-to-know. Essentially, this corresponds to a set of *user:ids*; however, the ADEPT implementation of Franchise is different for each security object:

1. User: All users wishing ADEPT service must be known to the system. This knowledge is imparted by SYSLOG at start-up time and limited to approximately 500 *user:ids* ($\max(U) \leq 500$).

2. Terminal: Equation (5) specifies the Franchise of a given terminal, $F_t$, as a set of *user:ids*. In ADEPT, $F_t$ does not exist. One may define all the users for a given terminal, i.e., $F_t$; or alternatively, all the terminals for a given user. Because SYSLOG orders its tables by *user:id*, the latter definition was found more convenient to implement.
3. Job: The Franchise of a job is the *user:id* of the creator of the job at the time of LOGIN to the system. Currently, only one user has access to (and control of) a job ($\mu = 0$ for equation (6)).
4. File: Implementation of Franchise for a file ($F_f$), is more extensive than equation (7). In ADEPT, we wish to control not only who accesses a file, but also the quality of access granted. We have defined a set of four exclusive qualities of access, such that a given quality, q, is defined if

$$q \in \{\text{READ, WRITE, READ-AND-WRITE, READ-AND-WRITE-WITH-LOCKOUT-OVERRIDE}\} \qquad (27)$$

ADEPT permits simultaneous access to a file by many jobs if the quality of access is for READ only. However, only one job may access a file with WRITE, or READ-AND-WRITE quality. ADEPT automatically locks out access to a file being written to avoid simultaneous reading and writing conflicts. A special access quality, however, does permit lockout override. Equation (7) can now be extended as a set of pairs,

$$F_f = \{(u_f^0, q^0), (u_f^1, q^1), \cdots, (u_f^\gamma, q^\gamma)\} \ ; \qquad (28)$$

where $q^i$ are not necessarily distinct and are given by equation (27).

The implementation of equation (28) is dependent upon $\gamma$, the number of franchised users. When $\gamma = 0$, we have the ADEPT Private file, exclusive to the owner, $u_f^0$; for $\gamma = \max(U)$, we have the Public file; values of $\gamma$ between these extremes yield the Semi-Private file. $\gamma$ is implicitly encoded as the ADEPT "privacy" item in the file's catalog control data, and takes the place of $F_f$ for all cases except a Semi-Private file. For that case exclusively, equation (28) holds and an actual $F_f$ list of *user:id, quality* pairs exists as a need-to-know list. The owner of a file specifies and controls the file's privacy, including the composition of the need-to-know list.

## Security control initialization: SYSLOG

SYSLOG is a component of the ADEPT initialization package responsible for arming the security controls. It operates as one of a number of system start-up options prior to the time when terminals are enabled. SYSLOG sets up the security profile data for *user:id* and *terminal:id*, i.e., the "given constants" of Table I.

SYSLOG creates or updates a highly sensitive system disc file, where each record corresponds to an authorized user. These records are constructed from a deck of cards consisting of separate data sets for *compartment* definitions, *terminal:id* classification, and *user:id* clearance. The dictionary of *compartment* definitions contains the less-than-9-character mnemonic for each member of the Category set. Data sets are formed from the card types shown in Table II. Use of *passwords* is described later in the LOGIN procedure.

An IDT card must exist for each authorized user; the PWD, DEV, SEC, and CAT card types are optional. Other card types are possible, but not germane to security control, e.g., ACT for accounting purposes. More than one PWD, DEV, and CAT card is acceptable up to the current maximum data limits (i.e., 64 *passwords*, 48 *terminal:ids*, and 16 *compartments*).

A variety of legality checks for proper data syntax, quantity, and order are provided. SYSLOG assumes the following default conditions when the corresponding card type is omitted from each data set:

| | |
|---|---|
| PWD | No *password* required |
| DEV | All *terminal:ids* authorized |
| SEC | A = UNCLASSIFIED |
| CAT | C = null (all zero mask) |

This gives the lowest user clearance as the default, while permitting convenient user access. Various options exist in SYSLOG to permit maintenance of the internal SYSLOG tables, including the replacement or deletion of existing data sets in total or in part.

The sensitivity of the information in the security control deck is obvious. Procedures have been developed at each installation that give the function of deck creation, control, and loading to specially cleared security personnel. The internal SYSLOG file itself is protected in a special manner described later.

## Access control

A fundamental security concern in multi-access sysis that many users with different clearances will be simultaneously using the system, thereby raising the

TABLE II—SYSLOG control cards

| *Card Type* | *Purpose* |
|---|---|
| DICT | Identifies start of data set of *compartment* definitions. |
| *compartment*$_1$ ··· *compartment*$_{16}$ | Defines up to 16 *compartments*. |
| TERMINAL | Identifies start of data sets of terminal definitions. |
| UNIT *terminal:id* | Identifies start of a terminal data set. |
| IDT *user:id* | Identifies start of a user data set. |
| PWD *password* ··· *password* | Defines legal *passwords* for *user:id* up to 64. |
| DEV *terminal:id*$_1$ ··· *terminal:id*$_{48}$ | Defines legal terminals for *user:id* up to 48. |
| SEC *Authority* | Defines *user:id* Authority. |
| CAT *compartment*$_1$ ··· *compartment*$_{16}$ | Defines *user:id* Category set. |

possibility of security compromise. Since programs are the "active agents" of the user, the system must maintain the integrity of each and of itself from accidental and/or deliberate intrusion. A multifile system must permit concurrent access by one or more jobs to one or more on-line, independently classified files.

ADEPT is all these things—multiuser, multiprogram, and multifile system. Thus, this section deals with access control over users, programs, and files.

## User access control: LOGIN

To gain admittance to the system, a user must first satisfy the ADEPT LOGIN decision procedure. This procedure attempts to authenticate the user in a fashion analogous to challenge-response practices.

The syntax of the ADEPT LOGIN command, typed by a user on his terminal, is as follows:

/LOGIN *user:id password accounting*

Figure 1 pictorially displays the LOGIN decision procedure based upon the user-specified input parameters. *User:id* is the index into the SYSLOG file used to retrieve the user security profile. If no such record exists (i.e., equation (22) fails), the LOGIN is unsuccessful and system access is denied. If the security profile is found, LOGIN next retrieves the *terminal:id* for the keyboard in use from internal system tables, and searches for a match in the *terminal:id* list for which the *user:id* was franchised by SYSLOG. An unsuccessful search is an unsuccessful LOGIN.

If the terminal is franchised, then the current *password* is retrieved from the SYSLOG file for this *user:id* and matched against the *password* entered as a keyboard parameter to LOGIN. An unsuccessful match is again an unsuccessful LOGIN. Furthermore, the terminal is ignored (will not honor input) for approximately 30 seconds to frustrate high-speed, computer-assisted, penetration attempts. If, however, the match is successful (equation (22) holds), the current *password* in the SYSLOG file for this *user:id* is discarded and LOGIN proceeds to create the job clearance.

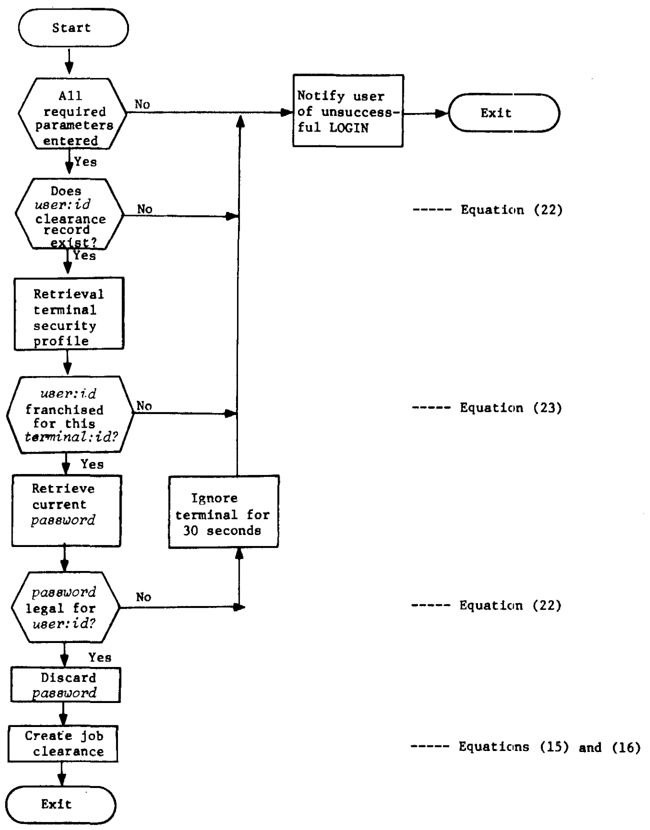

Start
All required parameters entered
No
Notify user of unsuccess-ful LOGIN
Exit
Yes
Does user:id clearance record exist?
No
----- Equation (22)
Yes
Retrieval terminal security profile
user:id franchised for this terminal:id?
No
----- Equation (23)
Yes
Retrieve current password
Ignore terminal for 30 seconds
password legal for user:id?
No
----- Equation (22)
Yes
Discard password
Create job clearance
----- Equations (15) and (16)
Exit

Figure 1—LOGIN decision procedure

*Passwords* in ADEPT obey the same syntax conventions as *user:id.* (See the earlier description of User Identification.) Although easily increased, currently SYSLOG permits up to 64 *passwords*. Each successful LOGIN throws away the user *password*; 64 successful LOGINs are possible before a new set of *passwords* need be established. If other than random, once-only *passwords* are desired, the 64 *passwords* may be encoded in some algorithmic manner, or replicated some number of times. Once-only *passwords* is an easily implemented technique for user authentication, which has been advocated by others.[2,7] It is a highly effective and secure technique because of the high permutability of 12-character-*passwords* and their time and order interdependence, known only to the user.

Once the authentication process is completely satisfied, LOGIN creates the job security profile according to equations (15) and (16) of our model. That is, the lower Authority of the user and the terminal becomes $A_j$, and the intersection (logical AND) of the user and terminal Category sets becomes the Category of the job, $C_j$. For example, a user with TOP SECRET Authority and a Category set (1001 1001 0000 1101) operating from a SECRET level terminal with a Category set (0000 0000 0000 0010) controls a job cleared to SECRET with an empty Category set.

## Program access control: LOAD

As noted earlier, the ADEPT Executive consists of two parts: BASEX, the resident part, and EXEX, the swapped part. EXEX is a body of reentrant code shared by all users; however, it is treated as a distinct program in each user's job. Up to four programs can exist concurrently in the job. Each operates with the job clearance—the job clearance umbrella.

LOAD is the ADEPT component used to load the programs chosen by the user; it is part of EXEX and hence operates as part of the user's job with the job's clearance. Programs are cataloged files and as such may be classified with a given security profile. As is described in "File Access Control" below, LOAD can only load those programs for which the job clearance is sufficient. Once loaded, however, the new program operates with the job clearance.

In this manner, we see the power of the job umbrella in providing smooth, flexible user operation concurrent with necessary security control. Program files may be classified with a variety of security profiles and then operate with yet another, i.e., the job clearance. By this technique security is assured and programs of different classifications may be operated by a user as one job. It permits, for example, an unclassified program file (e.g., a file editor) to be loaded into a highly classified job to process sensitive classified data files.

## File access control: OPEN

Before input/output can be performed on a file, a program must first acquire the file by an OPEN call to the Cataloger. Each program must OPEN a file for itself before it can manipulate the file, even if the file is already OPENed for another program. A successful OPEN requires proper specification of the file's descriptors—some of which are in the OPEN call, others of which are picked up directly by the Cataloger from the job environment area (e.g., job clearance, *user:id*)—and satisfactory job clearance and *user:id* need-to-know qualifications according to equations (25) and (26) of our model. Equation (25) is implemented as (8) as a straightforward magnitude comparison between $A_j$ and $A_f$. Equation (25) is implemented as (9) as an equality test between $C_f$ and $(C_j \wedge C_f)$. We use $(C_j \wedge C_f)$ to ensure that $C_f$ is a subset of the job categories; i.e., the job umbrella. Lastly, equation (26) is a NOP if the file is Public; a simple equality test between $u_j$ and $u_f$ if the File is Private; and a table search of $F_f$ for $u_j$ if the file is Semi-Private. These tests do increase processing time for file access; however, the tests are performed only once at OPEN time, where the cost is insignificant relative to the I/O processing subsequently performed on the file.

The quality of access granted by a successful OPEN, and subsequently enforced for all I/O transfers, is that requested, even if the user has a greater Franchise. For example, during program debugging, the owner of a file may OPEN it for READ access only, even though READ-AND-WRITE access quality is permitted. He thereby protects his file from possible uncontrolled modification by an erroneous WRITE call.

Considerable controversy surrounds the issue of automatic classification of new files formed by subset or merger of existing files. The heart of the issue is the poor accuracy of many such classification techniques[17] and the fear of too many over-classified files (a fear of operations personnel) or of too many under-classified files (a fear of the security control officers). ADEPT finesses the problem with a clever heuristic—most new files are created from existing files, hence classify the new file as a private file with the composite Authority and Category of all files referenced. This is achieved in ADEPT by use of the "high-water mark."

Starting with the boundary conditions of equations (11) and (13), the Cataloger applies equations (12) and

(14) for each successful file OPEN, and hence maintains the composite classification history of all files referenced by the job. For each new and temporary file OPEN, the Cataloger applies equations (17), (18), and (19); they are reapplied for each CLOSE of a new file, to update the classification (due to changes in the high-water mark since the OPEN) when the file becomes an existing cataloged file in the inventory. The scheme rarely underclassifies, and tends to overclassify when the new file is created late in the job cycle, as shown by boundary equations (20) and (21).

*Trans-formal security features*

ADEPT contains a host of features that transcend the formalism presented earlier. They are described here because they are integral to the total security control system and form a body of experience from which new formalisms can draw.

## Computer hardware

ADEPT operates on an IBM System 360/50 and is, therefore, limited to the hardware available. Studies by Bingham[9] suggest a variety of hardware features for security control, many of which are possessed by System 360.

IBM System 360 can operate in one of two states: the Supervisor state, or the Problem state. ADEPT executive programs operate in the Supervisor state; user programs operate in the Problem state.

A number of machine instructions are "privileged" to the Supervisor state only. An attempt to execute them in the Problem state is trapped by the hardware and control is returned to the executive program for remedial action. ADEPT disposes of these alarms by suspending the guilty job. (A suspended job may be resumed by the user.) Clearly, instructions that change the machine state are privileged to the executive only.

Another class of privileged instructions consists of those dealing with input/output. Problem state programs cannot directly access information files on secondary memory storage devices such as disc, tape, or drum. They must access these files indirectly by requests to the executive system. The requests are subjected to interpretive screening by the executive software.

Main memory is selectively protected against unauthorized change (write protected). We have also had the 360/50 modified to include fetch protection, which guards against unauthorized reading of—or executing from—protected memory. The memory protect instructions are also privileged only in the Supervisor state.

ADEPT software protects memory on a 4096-byte "page" basis (the hardware permits 2048-byte pages), allowing a non-contiguous mosaic of protected pages in memory for a given program. To satisfy multiprogramming, many different protection groups are needed. Through the use of programmable 4-bit hardware masks, up to 15 different protection groups can be accommodated in core concurrently. ADEPT executive programs operate with the all-zero "master key" mask, permitting universal access by all Basic and Extended Executive components.

There are five classes of interrupts processed by System/360 hardware: input/output, program, supervisor call, external, and machine check. Any interrupts that occur in the Problem state cause an automatic hardware switch to the Supervisor state, with CPU control flowing to the appropriate ADEPT executive interrupt controller. All security-vulnerable functions including hardware errors, external timer and keyboard actions, user program service requests, illegal instructions, memory protect violations, and input/output, are called to the attention of ADEPT by the System/360 interrupt system. The burden for security integrity is then one for ADEPT software.

## Monitor software

Inducing the system to violate its own protection mechanisms is one of the most likely ways of breaking a multi-access system. Those system components that perform tasks in response to user or program requests are most susceptible to such seduction.

### On-Line debugging

The debugging program provides an on-line capability for the professional programmer to dynamically look at and change selected portions of his program's memory. DEBUG can be directed to access sensitive core memory that would not be trapped by memory protection, since, as an EXEX component operating in the Supervisor state, DEBUG operates with the memory protection master key. To close this "trap door," DEBUG always performs interpretive checks on the legality of the debugging request. These checks are based upon address-out-of-bounds criteria, i.e., the requested debugging address must lie within the user's program area. If not, the request will be denied and the user warned, but he will not be terminated as has been suggested.[7]

## Input/output

Input/output in System/360 is handled by a number of special-purpose processors, called Selector Channels. To initiate any I/O, it is necessary for a channel program to be executed by the Selector Channel.

SPAM, the BASEX component that permits symbolic input/output calls from user programs, is really a special-purpose compiler that produces I/O channel programs from the SPAM calls. These channel progams are subsequently delivered and executed by the ADEPT Input/Output Supervisor, IOS.

SPAM permits a variety of calls to read, write, alter, search for, and position to records within cataloged files. To achieve these ends, SPAM depends upon a variety of control tables dynamically created by the Cataloger in the job environment.

The initiating and subsequent monitoring of channel program execution is the responsibility of the BASEX Input/Output Supervisor, IOS. IOS is called to execute a channel program (EXCP). System components, such as SPAM, branch to IOS at a known entry point that is fetch-protected against entry in the Problem state. IOS is off-limits to user programs attempting to access cataloged storage. For protection against unauthorized EXCP requests, IOS always performs legality checks before executing a channel program. These checks begin by examination of the device addressed by the channel program. If it is the device address for cataloged storage, further checks are made to determine the machine state of the calling program. That state must be Supervisor state for the call to be honored. A call in the Problem state would indicate an illegal EXCP call from a user program.

IOS m. k [illegible] ther checks to guarantee the validity of an I/C request [illegible] checks to see that the specified buffer areas for the [illegible] transfer do not overlay the channel program itself, an [illegible] lie within the user's program memory area, i.e., do not modify or access system or protected memory.

Covert I/O violations are also forestalled since I/O components take direction from information stored in the job environment—an area read- and write-protected from Problem state programs.

## Classified residue

Classified residue is classified information (either code or data) left behind in memory (i.e., core, drum, or disc) after the program that referenced it has been dismissed, swapped out, or quit from the system. The standard solution to the problem is to dynamically purge the contaminated memory (e.g., overwrite with random numbers, or zeros). In a system supporting over ¼ billion bytes of memory, that solution is unreasonable and in conflict with high performance goals. ADEPT's solution to the dilemma of denying access to classified residue while maintaining high performance depends upon techniques of controlled memory allocation.

1. *Core Residue*

   As noted earlier, all core storage is allocated as 4096-byte pages. These pages are always cleared to zero when allocated, thereby overwriting any potential residue.

   Via the program's page map, the ADEPT executive system labels all code and data pages (they need not be contiguous) belonging to a given program with a single hardware memory protection key, thereby prohibiting unauthorized reading or writing by other, potentially co-resident user programs that may be in execution. Furthermore, BASEX keeps a running account of the status and disposition of all pages of core.

   The Loader and Swapper components of ADEPT always work with full 4096-byte pages. Unfilled portions of pages at load time are kept cleared to zero as when they were allocated, and the full 4096 bytes are swapped into core, if not already resident, each scheduled time slice. Further, newly allocated pages are marked as "changed" pages, thus guaranteeing subsequent swap out to drum.

   With these procedures, ADEPT denies access by a user or program to those pages of core not identified as part of his program, and clears core residue by over-writing accessible core at load and swap times.

2. *Drum Residue*

   ADEPT always clears a drum page to zero before it is allocated. The page may subsequently be cleared again to user-specified data. ADEPT also maintains a drum map that notes the disposition of all drum pages (800 pages for the IBM 2303 drum). Drum input/output, like all ADEPT I/O, is controlled by executive privileged instructions.

3. *Disc Residue*

   Disc files in ADEPT are maintained as "dirty" memory. That is, the large capacity of the file system makes it infeasible to consider automatic over-writing techniques for residue control; therefore, deleted disc tracks are returned to the available storage pool contaminated and unclean. It then becomes the burden of the

ADEPT file system to control any unauthorized file access, whether to cataloged files or uncataloged disc memory.

Team work between the Cataloger, SPAM and IOS components of ADEPT achieves this control via legality checking of all OPEN and I/O requests.

For example, all disc packs are labeled internally and externally with their *volume:id*, and this label is checked at the time of mounting by the Cataloger OPEN procedure to assure proper volume mounting. Tapes may also be labeled and checked as a user option.

Of particular note, SPAM always assumes that an end-of-file (EOF) immediately follows the last record written in a new file, and it prohibits reading beyond that EOF. Contaminated tracks allocated to new files cannot be read until they are first written. The act of writing advances the EOF and the user simultaneously over-writes the classified residue with his own data. The user cannot skip over the EOF, and the EOF location is itself protected in the job environment area.

4. *Tape Residue*
   No special features for tape residue control are implemented in ADEPT. Tape residue control is easily satisfied by manual, off-line tape degaussing prior to ADEPT use.

System files

Equation (28) led us to examine Private, Semi-Private, and Public files. ADEPT possesses two additional file privacies that transcend our model; both are system files. Privacy-4 system files are the need-to-know lists created by the Cataloger itself for Semi-Private files. Privacy-5 system files are private system memory for the SYSLOG files and the catalogs themselves.

Access to these files is restricted to the system only. Special access checks are made that differ from those of equations (25) and (26). First, a special *user:id* is required that is not a member of $U$ (i.e., not in the SYSLOG file). Second, the program making the OPEN call must be in Supervisor state. Third, the program making the OPEN call must be a member of a short list of EXEX programs. The list is built into the Cataloger at the time of compilation. In this manner, access to system files is severely restricted, even to system programs.

Security service commands

ADEPT provides a variety of service commands that involve security control. The commands are listed in Table III. Note that commands VARYON, VARYOFF, REPLACE, LISTU, AUDIT, AUDOFF, and WRAP-UP are restricted to a particular terminal—the Security Officer's Station.

TABLE III—Security service commands

| *Command* | *Purpose* |
|---|---|
| AUDIT* | Turns on security audit recording. |
| AUDOFF* | Turns off security audit recording. |
| CHANGE | Enables the owner of a file to change any of the access control information of the file. |
| CREATE | Enables a user to create a Semi-Private file and its need-to-know list. |
| LISTU* | Lists by *terminal:id* all the current logged in *user:ids.* |
| RECLASS | Enables a user to raise or lower his job clearance between the bounds of the original LOGIN and current high-water mark clearance. |
| RELOG | Like LOGIN, but reconnects a user to an already existing job, as when a remote terminal drops off the communications line. |
| REPLACE* | Enables a user to move his job to another terminal or to reclassify a given device. |
| SECURITY | Print on the user's terminal approximately every 100 lines (or only by requestd the job high-water mark (or clearance by request) as a reminder to the user an) as a classification stamp of the level of current security activity. |
| VARYON/VARYOFF* | Permits terminals to be varied on- and off-line for flexibility in system maintenance and configuration control. |
| WRAPUP* | Shuts down system after a specified elapsed time. |

* Restricted to Security Officer's Station only.

## Audit

The AUDIT function records certain transactions relating to files, terminals, and users, and is the electronic equivalent of manual security accountability logs. Its purpose is to provide a record of user access in order to determine whether security violations have occurred and the extent to which secure data has been compromised. The AUDIT function may be initiated only at start-up time, but may be terminated at any time. All data re recorded on disc or tape in real time so the data is safe if the system malfunctions. An auxiliary utility program, AUDLIST, may be used to list the AUDIT file. The information recorded is shown in Table IV.

Implementation of AUDIT is quite straightforward, a product of general ADEPT recording and instrumentation.[18,19] AUDIT is an EXEX component that is called by, and at the completion of, each function o be recorded. The information to be recorded is pass d to AUDIT in the general registers. Additional I/O overhead is the primary cost incurred in the operation of AUDIT, for swapping and file maintenance. This cost is nominal, however, amounting to less than one percent of the CPU time.

## SUMMARY

In summary we may ask: How well have we met our goals? First, we believe we have developed and success-

TABLE IV—Security events and information audited by ADEPT-50

| EVENT | TIME | STATUS | JOB SECURITY PROFILE | USER SECURITY PROFILE | ACCOUNT NUMBER | USER:ID | TERMINAL:ID | CPU TIME | NEW TERMINAL | TERMINAL SECURITY PROFILE | FILE NAME | FILE OWNER ID | FILE SECURITY PROFILE | FILE FORM | FILE VOLUME NUMBER | PROSE CATEGORY NAMES |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LOGIN | X | X | X | X | X | X | X | | | | | | | | | |
| LOGOUT | X | X | | | | | | X | | | | | | | | |
| OPEN FILE | X | X | | | | | | | | | X | X | X | X | X | |
| REOPEN[1] FILE | X | X | | | | | | | | | X | X | X | X | X | |
| CHANGE FILE | X | X | | | | | | | | | X | X | X | X | X | |
| CLOSE FILE | X | X | | | | | | | | | X | X | | X | | |
| DELETE FILE | X | X | | | | | | | | | X | X | | X | | |
| RECLASS | X | X | X | | | | | | | | | | | | | |
| REPLACE | X | X | | | | | X | | X | X | | | | | | |
| DEVICE LIST[2] | X | | | | | | | | | X | | | | | | |
| CATEGORY DICTIONARY[3] | X | | | | | | | | | | | | | | | X |
| RESTART[4] | X | | | | | | | | | | | | | | | |
| WRAPUP[5] | X | | | | | | | | | | | | | | | |

[1] This is the "OPEN existing file" command.

[2] A list of all the terminal devices and their assigned security and categories is recorded at each system load.

[3] A list of the prose category names is recorded at each system load.

[4] Whenever the system is restarted on the same day (and AUDIT had been turned on earlier that day) the time of the restart is recorded.

[5] The time that the AUDOFF action was taken, or the time that the WRAPUP function called AUDIT, to terminate the AUDIT function.

fully demonstrated a security control mechanism that more than adequately supports heterogeneous levels and types of classification. Of note in this regard is the LOGIN decision procedure, access control tests, job umbrella, high-water mark, and audit trails recording. The approach can be improved in the direction of more compartments (on the order of 1000 or more), extension of the model to include system files, and the implementation of a single Franchise test for all security objects. The implementation needs redundant encoding and error detection of security profile data to increase confidence in the system—though we have not ourselves experienced difficulty here. The increase in memory requirements to achieve these improvements may force numerical encoding of security data, particularly Category, as suggested by Peters.[7]

Second, SYSLOG has been highly successful in demonstrating the concept of "security arming" of the system at start-up time. Our greatest difficulty in this area has been with the human element—the computer operators—in preparing and handling the control deck. In opposition to Peters,[7] we believe the operator should not be "designed out of the operation as much as possible," but rather his capabilities should be upgraded to meet the greater levels of sophistication and responsibility required to operate a time-sharing system.[20] He should be considered part of line management. ADEPT is oriented in this direction and work now in progress is aimed at building a real-time security surveillance and operations station (SOS).

Third, we missed the target in our attempt to isolate and limit the amount of critical coding. Though much of the control mechanism is restricted to a few components—LOGIN, SYSLOG, CATALOGER, AUDIT—enough is sprinkled around in other areas to make it impossible to restrict the omnipotent capabilities of the monitor, e.g., to run EXEX in Problem state. Some additional design forethought could have avoided some of this dispersal, particularly the wide distribution in memory of system data and programs that set and use these data. The effect of this shortcoming is the need for considerably greater checkout time, and the lowered confidence in the system's integrity.

Lastly, on the brighter side, we were surprisingly frugal in the cost of implementing this security control mechanism. It took approximately five percent of our effort to design, code, and checkout the ADEPT security control features. The code represents about ten percent of the 50,000 instructions in the system. Though the code is widely distributed, SYSLOG, security commands, LOGIN, AUDIT, and the CATALOGER account for about 80 percent of it. The overhead cost of operating these controls is difficult to measure, but it is quite low, in the order of one or two percent of total CPU time for normal operation, excluding SYSLOG. (SYSLOG, of course, runs at card reader speed.) The most significant area of overhead is in the checking of I/O channel programs, where some 5 to 10 msec are expended per call (on the average). Since this time is overlapped with other I/O, only CPU bound programs suffer degredation. AUDIT recording also contributes to service call overhead. In actuality, the net operating cost of our security controls may be zero or possibly negative, since AUDIT recordings showed us numerous trivial ways to measurably lower system overhead.

## ACKNOWLEDGMENTS

I would like to acknowledge the considerable encouragement I received in the formative stages of the ADEPT security control design from Mr. Richard Cleaveland, of the Defense Communications Agency (DCA). I would like to thank Mrs. Martha Bleier, Mr. Peter Baker, and Mr. Arnold Karush for their patient care in designing and implementing much of the work I've described Also, I wish to thank Mr. Marvin Schaefer for assisting me in set theory notation. Finally, I would like to applaud the ADEPT system project personnel for designing and building a time-sharing system so amenable to the ideas discussed herein.

## REFERENCES

1 A HARRISON
*The problem of privacy in the computer age: An annotated bibliography*
RAND Corp Dec 1967 RM-5495-PR/RC

2 L J HOFFMAN
*Computers and privacy: A survey*
Stanford Linear Accelerator Center Stanford Univ Aug 1968 SLAC-PUB-479

3 H E PETERSEN R TURN
*System implications of information privacy*
Proc SJCC Vol 30 1967 291-300

4 W H WARE
*Security and privacy in computer systems*
Proc SJCC Vol 30 1967 279-282

5 W H WARE
*Security and privacy: Similarities and differences*
Proc SJCC Vol 30 1967 287-290

6 R LINDE C WEISSMAN C FOX
*The ADEPT-50 time-sharing system*
Proc FJCC Vol 35 1969 Also issued as SDC Doc SP-3344

7 B PETERS
*Security considerations in a multi-programmed computer system*
Proc SJCC Vol 30 1967 283-286

8 RYE CAPRI COINS OCTOPUS SADIE Systems

NOC Workshop National Security Agency Oct 1968
9 H W BINGHAM
*Security techniques for EDP of multi-level classified information*
Rome Air Development Center Dec 1965 RADC-TR-65-415
10 R M GRAHAM
*Protection in an information processing utility*
ACM Symposium on Operating Systems Principles Oct 1967 Gatlinburg Tenn
11 L J HOFFMAN
*Formularies—Program controlled privacy in large data bases*
Stanford Univ Working Paper Feb 1969
12 D K HSIAO
*A file system for a problem solving facility*
Dissertation in Electrical Engineering Univ of Pa 1968
13 J I SCHWARTZ C WEISSMAN
*The SDC time-sharing system revisited*
Proc ACM Conf 1967 263-271
14 P BARAN
*On distributed communications: IX, security, secrecy, and tamper-free considerations*
RAND Corp Aug 1964 RM-3765-PR
15 C WEISSMAN
*Programming protection: What do you want to pay?*
SDC Mag Vol 10 No 8 Aug 1967
16 J P TITUS
*Washington commentary—Security and privacy*
CACM Vol 10 No 6 June 1967 379-380
17 I ENGER et al
*Automatic security classification study*
Rome Air Development Center Oct 1967 RADC-TR-67-472
18 A KARUSH
*The computer system recording utility: Application and theory*
System Development Corp March 1969 SP-3303
19 A KARUSH
*Benchmark analysis of time-sharing systems: Methodology and results*
System Development Corp April 1969 SP-3343
20 R R LINDE P E CHANEY
*Operational management of time-sharing systems*
Proc 21st Nat ACM Conf 1966 149-159

# Management of confidential information

*by* EDWARD V. COMBER

*System Dynamics, Inc.*
Oakland, California

## INTRODUCTION

For many years, informed persons have expended considerable time and energy attempting to evolve an acceptable philosophic assessment of the concept of "privacy." Studies made in the fields of anthropology, phychology, and sociology are in general agreement that both the mental and physical well-being of an individual requires freedom to experience some degree of personal anonymity within the environment. While the significance of "privacy" has been recognized, it has eluded the constraint of an acceptable definition. The search for a workable definition continues as man seeks a means for establishing practical bounds for inter-personal relations.

Recently, the concern for "privacy" has become a rallying point for those who see the present growth and applications of data automation as a threat to the "rights of privacy" of the individual. These advocates lament that the individual is unaware of the threat to his "loss of privacy" as his attention is diverted by the glowing promises of anticipated benefits that may become available through data automation.

It is the writer's belief that through the proper and reasonable utilization of the tools of modern data technology man will have within his power a mechanism that has the potential of becoming his strongest ally in his search for means to preserve the values of "privacy." In reality, the critical element in this question of "privacy" should not address itself to the electromechanical capability of the computer or system telecommunications functions. The true focal point is the direct challenge to the discipline and conduct of man who is the designer and user of the data system.[5] Man must be willing to abide by the standards he derives from his own "privacy" criteria. He must staunchly forego any temptation to engage in system shortcuts, and he must hold to the position that he will not accept lightly any violations of the "confidentiality controls" established for system operation. Any breach in the integrity of the system must be viewed as a direct personal challenge to the integrity of each person associated with the undertaking.

## SUMMARY

The following is a brief resume of significant elements that have been identified with the question of "privacy." These comments are not offered as final nor are they to be considered as embracing the entire area of concern. The summary is presented simply as a means of bringing together some key factors that could serve as a foundation for a basic "privacy" control system. The working standards will evolve as man gains more experience with this powerful ally and is able to resolve philosophical and ethical questions that are inherent in the concept of "privacy". As the environment and pace of modern life adjust to current needs, the nature of "privacy" will probably also reflect changes in priorities and the character of the social stresses.

### *Elements in the invasion of privacy*

No definitive statement exists which provides a clear and acceptable statement of what is "private information," or what constitutes an "unwarranted invasion of privacy." Any criteria proposed to date to identify "private information," or describe an act that would constitute "unwarranted invasion of privacy," must take into account whether or not such disclosure of the specific data:

A. *Would relate to an individual*, a family or other small group in such manner as to facilitate the likelihood of the unwarranted identification of the individuals, *or*
B. *The data is not considered public information* by provision of legal statute, *or*
C. *Would cause or be the basis for unjust economic loss or social stigma* or harassment to the individual, *or*
D. *Result in the unnecessary loss of a property right.*

*What is private vs. what is confidential?*

When attempting to discuss "privacy," the term "confidentiality" inevitably will join the debate, but does not promote clarification. What sort of personal information do reasonable men interpret as "private?" The answer to this question depends upon many things; for example, any one or more of the following factors may apply:

A. *The context within which the specific information* is embedded,
B. *The amount of information* assembled and accessible,
C. *The intrinsic nature* of the information.
D. *The sophistication of the social values* held by the individuals concerned,
E. *The character and scope of the sub-culture,*
F. *Significance of personal attributes* such as: age, ancestry, social status, race, etc.

Recently, the California Intergovernmental Board on EDP was established by statute.[1] It is charged with responsibility to provide for intergovernmental representation in the coordination of the many government sponsored EDP programs and to take leadership in the establishment of intersystem standards. The Intergovernmental Board appointed a select Technical Advisory Committee to assist in the preparation of a Manual to serve as a guideline for all agencies in the development of local systems and facilitate adequate interface capability as required. The manual was completed and is under review by the Intergovernmental Board prior to general release to official agencies throughout the State of California.

A sub-committee of the Technical Advisory Committee was specifically assigned to address the question of "privacy". The members of the Privacy Sub-committee concluded, after some study, that there are a number of personal information items that could be made accessible to an integrated data system without any threat to the individual "privacy". It was also recognized that there are many other data items that for one reason or another should be restricted from wide access in the absence of an established right to know. Some examples of these data items are as shown below:

A. *Information that may not be relevant to personal privacy*:

| | |
|---|---|
| Name | Sex |
| Maiden Name | Marital Status |
| Address | Name of Spouse |
| Age or DOB | Next of Kin |
| Race | |

B. *Information that would probably be relevant to personal privacy*:

Occupation
Education
Income
Religious Preference
Political Preference
Family Size
Number of Children
Ages of Children
Taxes Paid
History of Residence
Attitudes Toward Social Issues
Property Ownership
Value of Real Property
Marital History
Drinking Practices
Hospitalization Record
Medical Record
Symptoms of Illness
Record of Arrest
Ancestry
Nationality
Name of Relatives
Response to Psychological
  or Medical Questions

*Proliferation of data items throughout culture*

While some of the information items mentioned above may be found on records that are classified as confidential, many of the information items may also be found on records that are not subject to restriction by law or policy. The current trend in social intercourse and information exchange reflects an ever-broadening depth of self-revealment by individuals. Private and governmental services are being extended into newer areas and thereby attracting the participation of an ever-growing segment of the citizenry.

The integration of interagency information systems with data exchange introduces a new dimension associated with the creation of composite record images of persons known to the total system. These images are the product of independent and frequently unrelated inputs of data to serve other specific needs. Any integrated interagency information system with this potential capability must be administered by professionally qualified persons who remain sensitive of the need to verify both the identification of the subject of inquiry and the inquirer's "right to know". As more data systems are activated and interfaces are established, the individual who is the initial source of the data becomes more remote and isolated from the operational inquiry that relates to his record. It should be the constant aim of the system design, operational programming, and user discipline to assure that system integrity is not subverted.

*Significance of developing standards for data verification*

Attention should not be directed solely to provide for the identification and classification of personal data items. What is equally important, standards must be developed and adopted to guide data acceptance and utilization with respect to the ability to verify the information. For example, the confidence in the operating system will be increased and utilization encouraged if the user is assured that data items are subject to verification as to:

A. *Accuracy*
B. *Bias*
C. *Completeness*
D. *Currency*
E. *Documentation*
F. *Satisfaction of Legal Requirements*

A safety value that will support a sound verification program is to initiate a practical data purge system. The best data system in terms of cost/benefit analysis is one that has a high content of active data and one that is adequately updated. The effect of establishing a continuous and critical purge system is to provide an orderly review of file content, to remove inactive or low value data.

*One approach to a data classification plan*

A number of studies have been undertaken in an attempt to identify and define data items that should be processed as classified or confidential. There have been perhaps as many solutions offered as there have been studies proposed. The Privacy Sub-committee mentioned above proposed a simple three category data plan for consideration and approval of the California Intergovernmental Board on EDP.[2] The concept is summarized below:

A. *Confidential*:

This classification has the highest level of restriction, and should be limited to data which is prohibited from free and full disclosure by statutory regulation (law).

B. *Restricted*:

This is data which:

1. Is not prohibited from full and free disclosure by statute (coufidential), *and*
2. An unauthorized intrusion could constitute an unwarranted invasion of personal privacy, *and*
3. Has been administratively assigned a security classification—restricted.

C. *Unclassified*:

All data maintained by a public agency not otherwise classified as confidential or restricted as defined below.

*Sources of classification criteria*

The criteria for the establishment of classification of data arise from a variety of sources. In many instances, the criteria is a result of the interaction of one or more of the following:

A. *Public Policy*:
The living residue of tradition and social acceptance.
B. *Statutory Law*:
The formalized and legal codification of social needs and standards of conduct.
C. *Legal Interpretation*:
The implementation of judicial and administrative decisions that have been sanctioned through public acceptance.
D. *User Agency Specifications*:
Operational decisions that have been adopted

and ennunciated to promote agency goals in an atmosphere of public support.

E. *Personal Needs of The Individual*:
Acceptance of the system integrity by the public who participate and furnish personal information to assist an agency function with respect to the needs of the individual (Federal Census, Social Security, etc.).

Each of the sources of criteria utilized is subject to its own characteristic variations, and will require continuous reevaluation. The scope of data items subject to the confidential classification are under constant adjustment and reassessment due to the dynamic character of the social conditions which give rise to the data.

*Identification of areas sensitive to intrusion*[3]

One of the main deterrents to the development of new ideas about privacy has been the lack of specificity as to where the threats to privacy may arise. Many agree that at some future date, a serious threat may develop. That a real danger exists today is not universally accepted.

Let us consider the potential challenge to "privacy" that may originate from any of the following sources:

A. *The accidental observance of data by an individual.*
B. *The accidental dumping of a volume of confidential data to general view.*
C. *The solitary snoop.*
D. *The snoop-for-pay* (hired spy).
E. *The file stealer.*
F. *Misuse of confidential file by administrator* having access to system.
G. *Organized crime.*
H. *Totalitarian government.*
I. *Another possibility might be the intrusion of the private sector into government data files.*

*Establish policy on data classification*

Before any acceptable automation program can be developed to process information that may be considered "private" or "confidential," certain policy decisions must be resolved.

A. *The responsible administrators representing users of the system must reach agreement on the data content of the information system.* This agreement must include the identification of any data items or files that would be subject to restricted access or inquiry. If the restriction is pursuant to current policy, said policy should be specified:

1. General Public Policy
2. Agency Administrative Policy
3. Statutory Provision
4. Judicial Ruling

B. *Specific criteria should be established based on the accepted policy statements,* and serve as a guide to test the classification of all data introduced into the system. The continued validity of a classification should be based upon periodic challenge and justification.

C. *A policy manual should be prepared and maintained as a ready* reference to facilitate system operation.

1. Personnel participating in the system should be held individually accountable for full compliance with the "policy guidelines."
2. The policy manual should be subject to continuous review and update to remain current with system requirements, technology, and legal specifications.

D. *Additional considerations* in the development of an Interagency Information System to maintain privacy control. Decisions regarding the following elements of the system design and operation will prove significant:

1. *Facility Security*:
    (a) Location of Hardware
        Single *vs.* Multiple Facility
    (b) Physical Adequacy
        Equipment
        Personnel
    (c) Access to Facility
        Normal
        Emergency

2. *Equipment*:
    (a) Selection
    (b) Configuration
    (c) Operating Characteristics
        Multi-processing
        Multi-programming
        Remote Terminals

3. *Program Control*:
    (a) Single Management Responsibility

User Representation and Participation
- (b) Operating System
  Monitor of System Services And Access
- (c) System Applications
- (d) Man Machine Interface (Key Consideration)
- (e) Modularization of System Applications
  Does Modularization Weaken Privacy Control?
- (f) Integration of Compatible Systems
  Does Program Control Reside With The Core System?

4. *The Human Factor*:
   This is the critical and perhaps most unpredictable element in the functioning process.
   - (a) Personnel Recruitment, Selection And Appointment
   - (b) Personnel Training And Supervision
   - (c) Maintenance of Operating Discipline
   - (d) Personnel Retention

*Precautions to minimize potential for "privacy" violations*

The same versatility and power that makes the computer valuable as a data manipulator can be employed to monitor system services and support human supervision procedures. The operating information system should provide (assuming an adequate system analysis and design):

A. *A Sound Data Classification System*
   1. Specify data subject to restricted access and special protection.
   2. Provide for isolated storage of restricted data if necessary.
   3. Determine who has right to access to confidential data and under what operating conditions.
   4. User agency personnel should be certified for access by administration.

B. *Physical Conditions*:

   What levels of control should be imposed to promote system integrity and at the same time provide a functional environment that will encourage system utilization by the participants for which it was designed.

   1. *Equipment (system hardware)*:
      - (a) Location and physical security of equipment.
        - (1) Central Computer Installation
        - (2) Associated Peripheral Equipment
        - (3) Back-Up Facilities—Duplicate Files
      - (b) Remote terminal installations (I/O devices.)
      - (c) Circuit Security
   2. *System Configuration*
      - (a) Central Data Bank *vs.* Dispersed Data Bases
      - (b) Central Data File *vs.* Central Index Concept
      - (c) Central System Control *vs.* Remote Terminal Activation
        - (1) Restricted Terminal Operation
        - (2) Multiple Function Remote Terminal
   3. *Software System Support*—Programming must be developed with an awareness of the need for system integrity and data security. Provision must be made to provide control over basic software components, such as:
      - (a) Program Library
      - (b) Back-Up Documentation
      - (c) Diagnostic And Test Routines
      - (d) Continuous Coding of Update Schedules That Support The Identification Schemes Inherent to The Confidentiality Control Programs
      - (e) Transaction Monitor Logs Should Be Designed to Provide The Basis For Operational Supervision But Not Reveal The Location or Content of The Confidential Files Which Are Subject to Monitor Control

4. *Personnel Requirements*—If the system equipment and facilities justify particular planning to minimize the hazards to confidentiality, it is certain that consideration be given to the personnel who will function in the system. The scope of attention should extend through both the employees who perform the technical services associated with EDP, and the operating personnel of the agency for which the information system was developed. Despite all that has been said heretofore, the "key" to security of information rests with the individuals who have access to the data system. Our personnel planning should encompass many specific areas. The following relate most directly to physical factors:
   - (a) Personal Safety
     - (1) Area Accessibility
     - (2) Emergency Provisions
   - (b) Personal Accountability
     - (1) Identification Control Plan
       - (a) Access to Installation
       - (b) Access to Specific Work Areas
     - (2) Is the Plan Practical—Used?
   - (c) Conveniences And Necessities
     - (1) Are They Adequate?
     - (2) Are They Properly Located?
   - (d) What Special Precautions Are Warranted When Non-employee Personnel Are Permitted Access to The Installation Area?
     - (1) Equipment Maintenance
     - (2) Building Service Maintenance

C. *System Design Considerations*:

Control provided through specific programming techniques.

1. Limiting Terminal Access to The System—Programming
   - (a) Classification Schedule (*Data Level Control*)
     - (1) Terminal Identification
     - (2) Terminal Verification
     - (3) User Identification
     - (4) User Verification
     - (5) Call-Back Concept
   - (b) Restriction of Detail of Information in Response to Inquiry (*Data Item Control*)
     - (1) Refer to Index - Pointer to Source Data
     - (2) Status Indicator
     - (3) Advise Supervisory Station
       - (a) Secure Permission to Interrogate The Restricted File
       - (b) Receive Selected Response Through Monitor Agent
     - (4) Specific Limitation on Terminal Operation
       - (a) Data Input
       - (b) Data Manipulation
       - (c) Data Output
       - (d) Data Change or Update
       - (e) Data Purge
2. Establish A Monitor On All Terminal Action to Intercept and Identify unauthorized attempts to access the system.
   - (a) Identify Transmitting Terminal And Location
   - (b) Identify Terminal Operator(?)
   - (c) Identify Specific Nature of Restricted Access Attempt
   - (d) Provide For Supervisory Level Notification of The Attempt to Support Maintenance of System Discipline
   - (e) Abort The Unauthorized Attempt to Secure Data
3. Maintain audit review of selected files to

facilitate the orderly purge of files and to check levels of file activity

(a) Establish, as necessary, periodic file review procedures to challenge the continued "confidential" status of individual data items to assure conformity with system policy and user need
(b) Maintain necessary statistical measures of activity in restricted files to document operational policy decisions.
(c) Provide special test routines to challenge the confidentiality procedures and verify system functional integrity
(d) The Human Factor— The concern for confidentiality of data and file security eventually will focus on an assessment of problems that arise from the human element in the man-machine system. Despite the sophistication exercised in system analysis, design and implementation, specific recognition must be given to the fact that people participate in system operations.

*What about a future computer utility?*[4]

With the rapid and diverse growth of computer services and recognizing the intimate relation between hardware facilities, communication channels and the users of the systems, it is no accident that discussion should arise about the future establishment of a computer-communication utility. The need for such a service becomes more apparent as we see the introduction of time-sharing systems and the implementation of large integrated data services that support major regional and even statewide programs. The arguments pro and con the justification for a computer-communication utility are beyond the scope of this paper. However, the utility concept does provide the opportunity to propose several avenues of approach to improving the "privacy" control aspect in personal data systems. One of the recurring suggestions has been to establish a system of certification and licensing for persons directly involved with the design, installation, management, and the operation of data systems containing sensitive personal information. A second device that could prove of value would be to effect control through regulation of the computer-communication utility service.

## CONCLUSION

*The challenge of privacy control*

Violations of standards regarding confidentiality or privacy of information occur when particular items of personal data furnished to an information system for approved selective use are released to unauthorized persons or in a manner that jeopardizes expected system integrity.

A. *The Predominance of The Human Factor*

The integrity of any information system regarding confidentiality or invasion of privacy will eventually be resolved at the level of the human factor. Machines, data sets, file cabinets, index cards, tape drives, disk files, memory modules, computers, report registers—each of these devices is an inanimate object devised by man to receive, transfer, or hold information items made available to the system through human intervention. Data stored in these devices are significant only insofar as the output is meaningful to man, and subject to change or exposure by the action of an individual. Data stored in an inactive or inaccessible device without human interaction will not reveal information that would provide the basis for a violation of privacy. The relationship between man and his information system can be described as consisting of the following basic elements:

(1) Man conceives the system.
(2) Man builds the elements necessary to provide the system.
(3) Man organizes the elements and establishes a scheme of operation.
(4) Man gathers the data that he introduces into the system.
(5) Man activates the system.
(6) Man commands the resources of the system.
(7) Man utilizes the results of the system in his external contacts in society.

The consistent factor in the above summation is the predominant relationship of *man* to the *system*. Man is responsible for creation of the system, the input of information, the manipulation of that information, and the final disposition of the data produced or revealed by the system.

B. *Personnel Standards Are Necessary*

Due to the prime significance of the human element in the integrity of any data automated system, the programs must address the following problems in a forthright manner:

(1) Personnel standards must be established for all participants.
(2) All accepted personnel must be indoctrinated on a continuing basis regarding the system objectives, functions, operational responsibility, etc.
(3) Specific training must be provided regarding system participation and terminal operation.
(4) Each installation should have competent supervision and a plan of routine inspection of operations.
(5) Each agency participating in a larger shared system must be accountable for the performance and integrity of its representatives. It must also be responsible for the release of any system information that is received from a classified file.
(6) All personnel who have access to the system should be required to sign a voluntary statement acknowledging their individual responsibility to protect the integrity of the system and respect the confidentiality of classified data. This statement could be a factor in the initial as well as continued employment.[4]

The operating system must prove convenient and satisfactory to the user. It must provide an effective service with assurance as to its accuracy and adequacy. Outputs should be tailored to meet the user need under the circumstances of the inquiry. The efficiency of the system should discourage any user development or maintenance of alternate or substitute systems. The man-machine interface should be maintained through the use of simple, direct devices with a minimum requirement for coding progressive verification, etc. An automated data system should be so designed and supported that the user is free to direct his full attention to his prime functional responsibility.

The information system must be a viable and practical tool. It should function at the convenience of the user, with intelligible outputs consistent in time and content to satisfy the service requirement. Where a system requires specific security restrictions, these must be furnished and function without imposing any awkward limitation on the legitimate user of the system.

C. *Weak Policy And Discipline Results in An Inferior System*

Recent critics have voiced objection to the development of major data banks and interagency information sharing systems in government service. Their objection has been based, in part, on certain practices associated with private credit bureau operations. The lament, properly uttered, pointed to a lack of data control and exercise of discretion by a number of these private agencies. While the economic and social value of credit rating bureaus is readily admitted, the loose policies regarding "privacy of data" casts a shadow regarding the ability to maintain integrity in a major information system. I believe it is an unfortunate and improper inference to conclude that public information systems cannot protect the "privacy" of information due to questionable practices among some business organizations established to collect and merchandise private information for profit.

D. *Limitation of Data Access of Specific Authorization*

Suggestions have been made that an individual should specify the extent of utilization of personal information and then the system be required to conform to the intention expressed by the individual. This proposal sounds reasonable, but on further consideration, presents subsequent problems in data management, modification of data use authorization, etc., that demand thorough study.

E. *Individual Right of Inspection of Record - File Correction*

Perhaps one of the most practical approaches toward satisfaction of individual "right to privacy," and at the same time facilitate the availability of the maximum of information resources to solve social needs is to make provision so that the individual can inspect the system files that contain his personal data. The individual should also have means to seek correction of any data item that is in error and subject to bias interpretation.

F. *Develop Realistic Data Purge Policy*

Attention should be given to the development of basic guidelines regarding the longevity of data resident in a file or information system. The current trend is to collect and classify more and more data on more and more people. While hopefully most of the data will have social value, I am sure that a significant quantity will provide little benefit to the individual or the community. It is not too early to consider the need for sound purge criteria so that the data retained in an operating system will offer the highest potential return for the energy expended.

G. *Adequate Training Programs Must Be Developed And Employed For The EDP Staff And Personnel of The User Agency Who Have Occasion to Engage The Data System*

The content should include an introduction to system design concepts, the overall functions and data processing applications that are components of the system and a thorough instruction in terminal man-machine dialog. In addition, some attention should be given to explaining the service philosophy with particular attention to the rules regarding access to and utilization of any information from confidential or restricted files. The legal and moral issues must be clearly defined, and an understanding accepted by all who engage the system that a violation of the security code regarding restricted data may be sufficient grounds for removal from system participation or dismissal.

The training program must be viewed as a continuing support function with periodic refresher classes, problem sessions, review of privacy criteria, etc. It is most important that the agency administrators and key supervisory personnel become involved in this program, and not leave the system discipline task to the technical staff who are not equipped nor responsible for this duty.

H. *Despite much uncertainty and misgivings as to the effectiveness in terms of "privacy" control that will result from the imposition of a licensing scheme, such a potential mechanism will be the subject of more intense consideration with the passage of time.*

## REFERENCES

1 Intergovernmental Board on Electronic Data Processing created by statute passed by Legislature of the State of California. S B No 1100. This statute established under sections No 11710-11720 of the Government Code

2 File Security Procedures—Report by Sub-Committee on Privacy and Confidentiality of the Intergovernmental Board on Electronic Data Processing Oct 18 1969

3 Ibid

4 D E SCHWEINFURTH
*The coming computer utility—Laissez-Faire licensing or regulation?*
Computer Digest May 1968

5 A F WESTIN
*Privacy and freedom*
Atheneum New York 1967

6 Hearings Before a Sub-Committee on the Committee on Government Operations House of Representatives—89th Congress (Second Session) July 26 27 and 28 1966

7 System Development Corp "SDC Magazine" Vol 10 Nos 7 and 8 July Aug 1967 (This issue focussed on the question of computer privacy.)

# Some syntactic methods for specifying extendible programming languages

by VICTOR SCHNEIDER

*Purdue University*
Lafayette, Indiana

## *Model of translator system*

Our model of a programming-language translator system is represented schematically in the block diagram of Figure 1. This diagram divides the translator system into two components. The first component T is a translator program that reads in and translates the valid programs of some programming language L. The output of the translator is a subset T(L) of the intermediate language. The second component is a system M for executing the programs translated into the intermediate language. It will be seen that, in this intermediate language, the operators follow their operands in postfix (reverse polish) form, and they are relatively machine independent. In this paper, we will be mainly concerned with defining the operation of the translator component by specifying the input-output relationships of the translator for a particular programming language. These relationships will be described in a syntactic notation that is independent of the particular translation algorithm used for implementing the translator T.

The language that was chosen as an example for this paper is Wirth and Weber's EULER.[14] EULER is quite similar to ALGOL 60 in appearance and capabilities, and it has additional features found in the LISP list-processing language. The original EULER syntax was written to conform to the requirements of a precedence translation algorithm,[14] and contains a number of syntactic rules whose purpose is to facilitate construction of a precedence translator from these rules. Because of the presence of these stylized rules, it was decided to rewrite the EULER grammar into a more compact and transparent form than the one in which it originally appeared. An Irons-style notation[2,3] was used to specify the translation of this new EULER grammar.

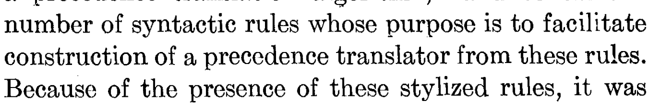


Figure 1—Simplified block diagram of a translator system

## *Reverse Polish translation of programming languages*

To illustrate what we mean by a syntactic specification of a programming–language translator, let us consider as an example the following small portion of the EULER syntax and examine some of the basic devices used by our EULER system:

*Grammar 1. A Simplified Subset of EULER*

| *Syntactic Rule* | *Rule of Translation* |
|---|---|
| ⟨expr⟩ → ⟨var⟩ = ⟨expr⟩ | ⟨var⟩⟨expr⟩ *assign* |
| \|⟨sum⟩ | I |
| ⟨sum⟩ → ⟨sum⟩ + ⟨term⟩ | ⟨sum⟩⟨term⟩*add* |
| \|⟨term⟩ | I |
| ⟨term⟩ → ⟨term⟩ * ⟨factor⟩ | ⟨term⟩⟨factor⟩*multiply* |
| \|⟨factor⟩ | I |
| ⟨factor⟩ → (⟨sum⟩) | ⟨sum⟩ |
| \|*at* ⟨var⟩ | ⟨var⟩ |
| \|⟨var⟩ | ⟨var⟩*in* |
| \|⟨var⟩. (⟨expr-sequence⟩). | ⟨expr-sequence⟩⟨var⟩*in* |
| ⟨var⟩ → ⟨name⟩ | *variable* ⟨name⟩ |
| ⟨expr-sequence⟩ → ⟨expr⟩ | I |
| \|⟨expr-sequence⟩, ⟨expr⟩ | ⟨expr-sequence⟩⟨expr⟩ |

Note that the rules of translation above refer to sequences of symbols on the right parts of syntactic rules. In this example, we see that the rules of translation specify how symbols and sequences of symbols in the source language are rearranged and rewritten in the translated language. Where no change at all is indicated in the translation of a particular rule, the symbol "I" appears as a translation rule. As an example of how sequences of symbols are rearranged for translation, the infix addition of

<sum> + <term>

is translated into the reverse polish sequence of symbols consisting of a "<sum>" followed by a "<term>" followed by the intermediate-language command for adding together the values resulting from evaluation of the previous two subexpressions. As in good polish notation, parenthesis are removed from around expressions, and this process is specified by associating the translation rule "<sum>" with the syntactic rule

<factor>→(<sum>).

The remaining rules having <factor> on the lefthand side are used for translating arithmetic operands into the intermediate language. For example, the syntactic rule

<factor>→<var>

indicates that operands in arithmetic expressions are variable names, and the translation of a <var> into the sequence

<var> *in*

indicates that the *"in"* command is used for fetching the value associated with <var> and for storing that value on top of the run-time operand stack of system M.

The other syntactic rule

<factor>→ *at* <var>

reflects the fact the EULER permits use of program variables that are pointers to data named by other program variables. Hence, the effect of the *"at"* command of the source language is to suppress the appearance of *"in"* in the translated program after the translated variable name. In this case, a pointer to the data stored in <var> is left on top of the operand stack in system M at run time. Finally, the rule

<var>→<name>

means that the names of program variables are translated into the sequence *"variable*<name>." Here, the effect of the *"variable"* command is to find a pointer to the data stored in the following name by system M and to place this pointer on top of the run-time operand stack.

The sequence "<var>.(<expr-sequence>)." on the right part of the remaining <factor> rule is the definition of an EULER function call. Function calls are translated with the parameters preceding the function name in the translated program. In this way, the function call can be made to look like a reverse polish operator having n operands, with n the number of parameters. A parameterless function call is translated exactly the same way as a program variable. Thus, the sequence

"*variable*<name>*in*"

in a translated program serves both to fetch data and to initiate a call on a function, depending on the <name> involved. This calling sequence will be referred to in the following discussion of extendible language features.

In the full translation grammar for EULER given in Appendix 2, it is possible to see how the methods presented in the preceding example are applied to the specification of a complete programming language. Note that this larger grammar uses, e.g., the symbol "+" in place of the "*add*" instruction of our small example, and, in general, translates as many source-language symbols as possible directly into commands of the intermediate language. The description of EULER programming given in Appendix 1 of this paper should clarify the meaning of the EULER operators used, and the following section in this paper will discuss the syntactic methods for optimizing and extending EULER as they are developed in the EULER grammar. A full description of the intermediate reverse-polish language specified by the EULER rules of translation can be found in Schneider.[10]

*Syntactic methods of optimizing expressions*

In the EULER grammar of Appendix 2, the rules of translation specify that a conditional statement or expression of the form

"IF <expr>$^1$ THEN <expr>$^2$ ELSE <-expr>$^3$"

is translated into its intermediate language version in the form

"<expr>$^1$ $IF <expr>$^2$ $THEN <expr>$^3$ $ELSE"

Note that each of the expressions here can themselves contain conditional expressions of any desired degree of nesting, and each of the subexpressions will be rearranged as shown above. In this intermediate language the "$IF" command causes an interpretive scan to the *matching* "$THEN" label if <expr>$^1$ is false. Otherwise execution continues until a "$THEN" is reached, at which point a scan occurs to the "$ELSE" label that matches this "$THEN". In this way, "$THEN" and "$ELSE" behave like balanced parentheses around expressions, and also serve as place-markers to which control can be transferred in the translated program.

This mechanism for executing translated cond tional expressions is used also as the basis for translating logical expressions into a partially optimized form. To take an example, the EULER sequence corresponding to a disjunction is represented by

"<disj> OR <conj>".

Its translated form is

"<disj> $IF $TRUE $THEN <conj> $ELSE".

Here, if the first operand "<disj>" of the expression is true, the entire expression is true. Therefore, the second operand is evaluated only if the first operand is false. A similar mechanism is used for the sequence

"<conj> AND <neg>".

Here, if the first operand is false, the second operand need not be evaluated. Hence, the translated conjunction is of the form

"<conj> $IF <neg> $THEN $FALSE $ELSE."

*Some syntactic methods of extending EULER*

After developing the appropriate techniques for translating conditional expressions and for optimizing logical expressions, the next order of business is to use these syntactic tricks to provide extended facilities in the EULER language. The introduction of full string-processing facilities into the EULER system is the first example to be considered. Without altering the EULER interpreter, and with a little reprogramming of the translator, we can effect the following improvement:

| *Syntactic Rule* | *Rule of Translation* |
|---|---|
| ⟨prim⟩ → ⟨stringprim⟩ | I |
| ⟨stringprim⟩ → ⟨stringhead⟩′ | ⟨stringhead⟩). |
| ⟨stringhead⟩ → ′ | |
| \| ⟨stringhead⟩⟨symbol⟩ | ⟨stringhead⟩.*⟨symbol⟩, |

Here, a string of arbitrary length is translated into a list whose cells store the symbols in the string one symbol in the cell in sequence. With this arrangement, it is possible to manipulate strings using the list concatenation operator provided by EULER, and using EULER subroutines to perform tests for list equality and containment.

The second example involves the addition of facilities for reading in data at run time within the framework of the EULER system. In this case, additional facilities must be provided in the EULER polish string interpreter. These facilities take the form of routines for converting numbers into their internal representation and for packing string data. The added syntax consists of the following set of rules:

| *Syntactic Rule* | *Rule of Translation* |
|---|---|
| ⟨program⟩ → .ENTRY ⟨block⟩.EXIT. | ⟨block⟩ |
| \|.ENTRY ⟨data⟩., ⟨block⟩ .EXIT. | ⟨data⟩⟨block⟩ |
| ⟨data⟩ → ⟨datahead⟩ END | I |
| ⟨datahead⟩ → DATA ⟨item⟩ | $DATA ⟨item⟩ |
| \|⟨datahead⟩., ⟨item⟩ | I |
| ⟨item⟩ → ⟨number⟩ | I |
| \|⟨stringprim⟩ | I |
| \|⟨datalist⟩ | I |
| ⟨datalist⟩ → .(). | I |
| \|⟨datalisthead⟩⟨item⟩). | I |
| ⟨datalisthead⟩ → .( | I |
| \|⟨datalisthead⟩⟨item⟩, | I |

With this program structure, the data portion could be read in by a run-time subroutine that leaves the data in a pre–arranged location of memory. The interpreter routine could then be read in over the data routine, and the translated program would be executed. A statement of the form "READ <prim>" would then store an appropriate link to some segment of the read-in <data> on top of the run-time operand stack.

The third example involves the use of a syntactic notation to expand the EULER language into a self-extendible programming language similar to MAD/1 (4) and ALGOL 68 (11). By an extendible programming language, people currently mean the following two things.

a. A language in which the programmer can specify new data types and data structures composed of novel configurations of data elements.
b. A language in which the programmer is able to reorder the priorities of expression operators and is able to specify arbitrary new operations at will.

In EULER, there already exists a general mechanism for allowing programmers to manipulate data structures, namely, the list mechanism. EULER lists can be constructed from arbitrary combinations of data elements. However, EULER only has eight data types with no facilities for extending their ranges. Such range-extension facilities depend on the machine on which the language is implemented, and algorithms for specifying such data types as numbers of arbitrary precision must be written for the machine in question. Hence, our example will concentrate on the *machine-independent problem* of specifying new operators in programs.

Any reasonable programming language must presuppose the existence of a standard set of expression operators before provision is made for allowing programs to expand this set of operators. With each standard operator will be associated a standard precedence level, and the operators to be introduced by the programmer must also have precedence levels. As the term is currently used, operator precedence (or priority) is a measure of how expression operators compare in binding power. For example, exponentiation is said to have lower precedence than addition, because exponentiation is performed before addition in arithmetic expressions. Thus, precedence imposes an ordering on the operations of a language. This ordering is reflected in the ordering of syntax rules in programming language grammars. In the EULER grammar above, rules are ordered so that list concatenation is performed first, then exponentiation, and so on, until the operation of value assignment. From concatenation to assignment of value there are nine levels of precedence.

Our approach in providing for the programming of new operations is to assign these operations to one of nine classes of operators, reflecting the nine levels in original grammar. This means that the translator must now treat operators as though they are procedure calls that can only be written into the translated program where their associated precedence level permits their operations to occur. In order to permit the programmer to tell the translator what precedence is associated with a newly defined operator, we require an additional operator declaration in our language. This declaration, together with the precedence syntax of expressions that follows, is sufficient to provide the expanded operator-definition facility

*Grammar 2. An Expression Grammar for Defining New Operators*

| *Syntactic Rule* | *Rule of Translation* |
|---|---|
| ⟨expr⟩ → ⟨var⟩⟨opname⟩⟨expr⟩ | ⟨var⟩⟨expr⟩ $VARBL ⟨opname⟩ $IN |
| \|⟨disj⟩ | I |
| ⟨disj⟩ → ⟨disj⟩⟨opname⟩⟨conj⟩ | ⟨disj⟩⟨conj⟩ $VARBL⟨opname⟩ $IN |
| \|⟨conj⟩ | I |
| ⟨conj⟩ → ⟨conj⟩⟨opname⟩⟨neg⟩ | ⟨conj⟩⟨neg⟩ $VARBL ⟨opname⟩ $IN |
| \|⟨neg⟩ | I |
| . | . |
| . | . |
| . | . |
| ⟨catena⟩ → ⟨catena⟩⟨opname⟩⟨prim⟩ | ⟨catena⟩⟨prim⟩ $VARBL ⟨opname⟩ $IN |
| \|⟨prim⟩ | I |
| . | . |
| . | . |
| . | . |
| ⟨blockhead⟩ → ⟨blockhead⟩ ⟨operatordec⟩., | ⟨blockhead⟩⟨operatordec⟩ |
| ⟨operatordec⟩ → OPERATOR ⟨opname⟩ | $NEW ⟨opname⟩ |
| \|⟨operatordec⟩, ⟨opname⟩ | ⟨operatordec⟩ $NEW ⟨opname⟩ |
| . | . |
| . | . |
| . | . |
| ⟨expr⟩ → ⟨opname⟩ = ⟨opdef⟩ | ⟨opname⟩⟨opdef⟩ = |
| ⟨opdef⟩ → ⟨defhead⟩⟨expr⟩ $. | I |
| ⟨defhead⟩ → ⟨rankpart⟩ ⟨operandpart⟩., | ⟨rankpart⟩⟨operandpart⟩ |
| ⟨rankpart⟩ → RANK OF ⟨digit⟩., | (Not Translated) |
| ⟨operandpart⟩ → OPERANDS ⟨name⟩ | $FORMA ⟨name⟩ |
| \|⟨operandpart⟩, ⟨name⟩ | $FORMA ⟨name⟩⟨operandpart⟩ |
| ⟨opname⟩ → ⟨symbol⟩ | I |
| \|⟨opname⟩⟨symbol⟩ | I |

In the expression syntax above, the <opname> in each rule is translated into a procedure call, with parameters consisting of the one or more operands associated with each <opname>. These procedure calls either refer to the "Standard" operator associated with a particular precedence level or refer to the translated <opdef> declared by the programmer. It is assumed that the translator will automatically enclose each translated program with an extra outer block containing procedure definitions for the set of standard operators basic to the language. In this way, the standard operators can be redefined within a particular program, but will regain their usual meaning upon exit from the block in which the redefining statement occurred. A consequence of this method of allowing new operator definitions is that program subroutines may use operators global to their definitions, but may not have operators passed to them as parameters, since all assignment of precedence is performed at translation time.

A certain amount of optimization is still possible within the framework of this extendible translator. As an example, suppose that we write the following procedure correspond to the standard operator for logical conjunction:

```
AND = RANK OF 7., OPERANDS X, Y., IF Y
THEN X ELSE FALSE $.
```

The actual parameters in the procedure call for logical AND above are expressions surrounded by ".$" and "$.". Thus, the effect of the conditional expression in the operator definition given above is to evaluate the Y parameter only once and not to evaluate the X parameter unless Y is true.

*Programmer-defined syntactic augments to existing languages*

As a next step in allowing programmers to decide on the nature of their own programming languages, we could conceive of a translator facility for allowing programmer-specified syntactic and semantic augments to existing programming languages. The idea behind this definitional facility is that the translator can be provided with facilities for accepting new syntactic rules and associating their right parts with rules of translation that are essentially calls on global procedures. The operands within the new syntactic augments are than translated as parameters supplied to the procedures for executing the augments. The feasibility of such augments, provided they do not lead to problems of syntactic ambiguity, can be inferred from the algorithms presented in Schneider.[9,10]

As an example of what a programmer might be tempted to add to his language, and of the methods he could use, we consider the problem of adding ALGOL W-style iteration to the EULER language. In the following translation grammar, the global procedures used in translated programs are "$FOR" and "$ WHILE", corresponding to the incremented variable and logical iterations, respectively.

*Grammar 3. A Programmer-Defined Syntax of Iterative Statements*

*Syntactic Rule*

(a) $\langle$expr$\rangle \rightarrow$ WHILE $\langle$expr$\rangle^1$ DO $\langle$expr$\rangle^2$
(b) $\langle$expr$\rangle \rightarrow$ FOR $\langle$var$\rangle$ FROM $\langle$expr$\rangle^1$ UNTIL $\langle$expr$\rangle^2$ BY $\langle$expr$\rangle^3$ DO $\langle$expr$\rangle^4$

*Rule of Translation*

(a) .$\langle$expr$\rangle^1$ \$..\$$\langle$expr$\rangle^2$ \$.\$VARBL \$WHILE \$IN
(b) $\langle$var$\rangle\langle$expr$\rangle^1$ $\langle$expr$\rangle^2$ $\langle$expr$\rangle^3$ .\$$\langle$expr$\rangle^4$ \$.\$VARBL \$FOR \$IN

Note that the controlled statement in the syntax above is translated with procedure definition brackets ".$." and "$.". In this way whenever the corresponding formal parameter in the "$FOR" OR "$WHILE" procedure definition is executed, the entire controlled statement is executed as a procedure. The procedure definitions of "$FOR" and "$WHILE" that follows are the "semantics" of Grammar 3:

```
$FOR = .$FORMAL VAR, EXP1, EXP2,
EXP3, STAT.,
   BEGIN LABEL TEST, CYCLE.,
      VAR = EXP1., GO TO TEST.,,
    CYCLE.. VAR = VAR+EXP2.,
    TEST.. IF(VAR-EXP3)*SIGN(EXP2)GT O
      THEN UNDEFINE D
      ELSE BEGIN STAT., GO TO CYCLE
      END $.

$WHILE = .$FORMAL LOGEXP, STAT.
  BEGIN LABEL CYCLE.,
  CYCLE.. IF LOGEXP THEN BEGINST AT,
      GO TO CYCLE END
      ELSE UNDEFINED END $.
```

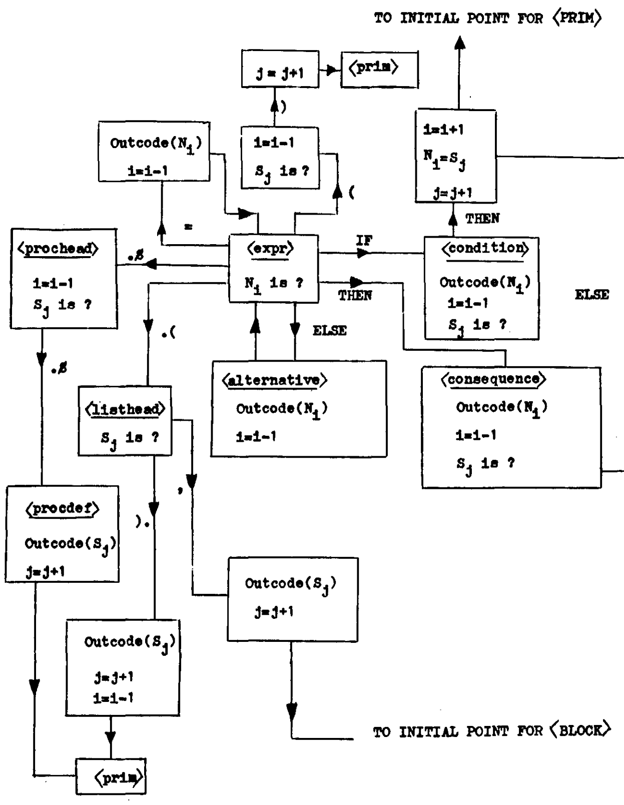

TO INITIAL POINT FOR ⟨PRIM⟩
j = j+1
⟨prim⟩
)
Outcode(N_i) i=i-1
i=i-1 S_j is ?
(
i=i+1 N_i=S_j j=j+1
THEN
=
⟨prochead⟩ i=i-1 S_j is ?
.ß
⟨expr⟩ N_i is ?
IF
⟨condition⟩ Outcode(N_i) i=i-1 S_j is ?
THEN
ELSE
.(
ELSE
⟨alternative⟩ Outcode(N_i) i=i-1
⟨consequence⟩ Outcode(N_i) i=i-1 S_j is ?
.ß
⟨listhead⟩ S_j is ?
,
⟨procdef⟩ Outcode(S_j) j=j+1
).
Outcode(S_j) j=j+1
Outcode(S_j) j=j+1 i=i-1
TO INITIAL POINT FOR ⟨BLOCK⟩
⟨prim⟩

Figure 2—A portion of the EULER translator

*How a section of the translator was designed—an example*

It is assumed that readers of this section will have some familiarity with the translator example in the previous paper[9] on this subject. In order to simplify the programming of the translator, it was decided to have the reserved words of the language perform as many functions as possible in the translation. Thus, the reserved words actually appear in translations as commands for the interpretive system where appropriate, and are stored on the pushdown store of the translator in place of the "nonterminal symbols" of the normal-form version of the grammar. For example, in the normal-form grammar for EULER, the rule for a conditional expression is

$$<\text{expr}> \rightarrow X_1 <\text{alternative}>$$

$$X_1 \rightarrow X_2 <\text{consequence}>$$

$$X_2 \rightarrow <\text{condition}>$$

By letting $X^1$ be THEN and $X^2$ be IF in the translator, the coding is greatly simplified, and no ambiguities are introduced, since the $X_i$ can be treated as "new and distinct" symbols of the normal-form grammar. The flowchart of Figure 2, showing the transitions to and from the box corresponding to <expr<, illustrates how the EULER translator was programmed.

## REFERENCES

1 R W FLOYD
*A descriptive language for symbol manipulation*
J ACM Vol 8 1961 579-584

2 E T IRONS
*A syntax directed compiler for ALGOL 60*
CACM Vol 4 1961 51-55

3 P M LEWIS R E STEARNS
*Syntax-directed transduction*
J ACM Vol 15 1968 465-488

4 D L MILLS
*The syntactic structure of MAD/1*
DDC Rpt No AD-671-683 1968

5 P NAUR editor
*Revised report on the algorithmic language ALGOL 60*
CACM Vol 6 1963 1-17

6 V B SCHNEIDER
*The design of processors for context-free languages*
N S F Memo Northwestern Univ 1965

7 V B SCHNEIDER
*Pushdown-store processors of context-free languages*
Dept of Industrial Engineering Northwestern Univ 1966
Evanston Ill

8 V B SCHNEIDER
*Syntax-checking and parsing of context-free languages by pushdown-store automata*
Proc SJCC 1967 71-75

9 V B SCHNEIDER
*A system for designing fast programming language translators*
Proc SJCC 1969 777-792

10 V B SCHNEIDER
*A translator system for the EULER programmng language*
Tech Rpt 68-76 Computer Science Center Univ of Md
College Park 1969

11 A VAN WIJNGAARDEN editor
*Report on the algorithmic language ALGOL 68*
Mathematisch Centrum 49 2e Boerhaavestraat Amsterdam
The Netherlands 1969

12 J WEIZENBAUM
*A symmetric list processor*
CACM Vol 6 1963 524

13 N WIRTH
*A generalization of ALGOL*
CACM Vol 6 1963 547-554

14 N WIRTH H WEBER
*A generalization of ALGOL and its formal definition: Parts I and II*
CACM Vol 9 1966 13-25 89-99

## *Appendix* I

*Features of the EULER language*

EULER is a nested block-structure language, similar to ALGOL. Thus, every block, consisting of a sequence of statements surrounded by BEGIN and

END parentheses, can be treated as a single statement in ALGOL fashion. An EULER program consists of an EULER block preceded by .ENTRY and followed by .EXIT..

In EULER, there are three declarations. One declaration is for data variables, one for program labels, and one for formal parameters of procedures. In the program

".ENTRY BEGIN NEW X, Y.,

LABEL Z., . . .

Z.. X + Y END .EXIT."

X and Y will store data, and Z will be a label preceding some statement.

Assigning a data type to a declared variable is accomplished by writing an assignment statement with data of the appropriate type on the right-hand side of the assignment. Thus, typing of variables in EULER is dynamic, since any assignment statement can change the data type stored in a variable. And, data typing is implicit, since there are no declarations like real, integer, etc., as appear in ALGOL. The following is a list of the right EULER data types:

I. Number —In the EULER system, all numbers are assumed to be floating point numbers. The assignment statement

"V = E.,"

with E a numerical expression or number, causes variable V to become a numerical variable.

II. Symbol —In this EULER implementation, an assignment statement such as

"V = .*ALPHAN.,"

causes the six characters "ALPHAN" to be stored in the location named by variable V.

III. Logical —The logical constants are TRUE and FALSE, standing respectively for logical truth and falsehood. The assignment statement,

"V = L.,"

with L a logical constant or logical expression, causes variable V to become a logical variable.

IV. Label —EULER programs use two declarations. "NEW" is used to declare a data variable, and "LABEL" is used to declare the presence of a label in some block of a program. Interestingly, if V is a variable in some EULER block, and V is not in a block global to the block of label L, then the assignment statement

"V = L.,"

causes V henceforth to be of type label, and to be interchangeable with L in GO TO statements.

V. Reference—In EULER, if VI is a variable not in a block global to the block of variable V2, then the assignment statement

"V1 = AT V2.,"

makes V1 a pointer to the data stored in V2. After V1 is turned into such a pointer, the two statements

"V2 = V2 + 1.,"
and "V1 IN = V1 IN + 1.,"

will have exactly the same effect of manipulating whatever data is stored in V2.

VI. Procedure—An assignment statement of the form

"V1 = .$ ⟨expr⟩ $..,"

causes V1 to become the name of a parameterless procedure call with body given by ⟨expr⟩. As a programming example, we might consider the following EULER block: "BEGIN NEW X, Y., X = 2.,

```
Y = .$FORMAL Z., X = X + Z$..,
OUT Y.(5). END"
```

When Y.(5). is operated on by the "OUT" operator, the value 7.0000 will be written out.

VII. List —In EULER, lists can be constructed in three distinct ways:
(a) On command: "V1 = LIST 300.,"
This statement creates a list of 300 undefined cells and makes V1 their name.
(b) By explicit notation: "V2 = .(1,.(2, 3)., 4)..,"
This statement creates a list consisting of two numbers and a sublist and makes V2 the name of that list.
(c) By concatenation: "V1 = V1 CONCAT V2.," Using the CONCATenation operator, small lists can be joined into larger ones.

In addition, lists can be subscripted in the same way as ALGOL arrays, each element of a list can be any EULER data type, including label, reference, and procedure. The following EULER block is a small example of the generality of the list notation: "BEGIN NEW X, Y., LABEL Z.,

```
Y = .(2, .$ BEGIN X = X + 1., Y(X) END $.,
    .$ OUTX$., Z)..,
X = Y(1)., Y(X)., GOTO Y(4).,
Z.. OUT .*FINISH END"
```

With this program segment, first 3.0000, then FINISH will be written out by the executed program.

VIII. Undefined—Every variable declared by "NEW" in an EULER program is initially of type "UNDEFINED." In addition, "UNDEFINED" is used as a data constant occasionally and as an empty option in conditional statements such as:

"V = IF L1 THEN .(1, 5). ELSE UNDEFINED.,"

For more details on EULER programming, the reader is referred to the Wirth and Weber EULER paper.[14]

*Appendix 2*

*A new translation grammar for EULER*

| | *Syntactic Rule* | *Rule of Translation* |
|---|---|---|
| 1: | ⟨program⟩ → .ENTRY ⟨block⟩ .EXIT. | ⟨block⟩ |
| 2: | ⟨block⟩ → ⟨blockhead⟩⟨body⟩ END | ⟨blockhead⟩⟨body⟩ $END |
| 3: | ⟨blockhead⟩ → BEGIN | $BEGIN |
| | \|⟨blockhead⟩⟨labeldec⟩., | ⟨blockhead⟩⟨labeldec⟩ |
| | \|⟨blockhead⟩⟨vardec⟩., | ⟨blockhead⟩⟨vardec⟩ |
| 4: | ⟨vardec⟩ → NEW ⟨name⟩ | $NEW name |
| | \|⟨vardec⟩, ⟨name⟩ | ⟨vardec⟩ $NEW ⟨name⟩ |
| 5: | ⟨labeldec⟩ → LABEL ⟨name⟩ | $LABEL ⟨name⟩ |
| | \|⟨labeldec⟩, ⟨name⟩ | ⟨labeldec⟩ $LABEL ⟨name⟩ |
| 6: | ⟨body⟩ → ⟨body⟩., ⟨stat⟩ | I |
| | \|⟨stat⟩ | I |
| 7: | ⟨stat⟩ → ⟨labdef⟩⟨stat⟩ | I |
| | \|⟨expr⟩ | I |
| 8: | ⟨labdef⟩ → ⟨name⟩.. | $LBDF ⟨name⟩ |
| 9: | ⟨expr⟩ GO TO ⟨expr⟩ | ⟨expr⟩ $GOTO |
| | \|OUT ⟨expr⟩ | ⟨expr⟩ $OUT |
| | \|⟨var⟩ = ⟨expr⟩ | ⟨var⟩⟨expr⟩ = |
| | \|⟨disj⟩ | I |
| | \|⟨condition⟩⟨consequence⟩⟨alternative⟩ | I |
| 10: | ⟨condition⟩ → IF ⟨expr⟩ | ⟨expr⟩ $IF |
| 11: | ⟨consequence⟩ → THEN ⟨expr⟩ | ⟨expr⟩ $THEN |
| 12: | ⟨alternative⟩ → ELSE ⟨expr⟩ | ⟨expr⟩ $ELSE |
| 13: | ⟨disj⟩ → ⟨conj⟩ | I |
| | \|⟨disj⟩ OR ⟨conj⟩ | ⟨disj⟩ $IF_$TRUE $THEN_ ⟨conj⟩ $ELSE |
| 14: | ⟨conj⟩ → ⟨neg⟩ | I |
| | \|⟨conj⟩ AND ⟨neg⟩ | ⟨conj⟩ $IF_⟨neg⟩ $THEN_ $FALSE $ELSE |
| 15: | ⟨neg⟩ → ⟨relation⟩ | I |
| | \|NOT ⟨relation⟩ | ⟨relation⟩ $NOT |
| 16: | ⟨relation⟩ → ⟨sum⟩ | I |
| | \|⟨sum⟩$^1$⟨relop⟩⟨sum⟩$^2$ | ⟨sum⟩$^1$⟨sum⟩$^2$⟨relop⟩ |
| 17: | ⟨relop⟩ → EQ\|NEQ\|GEQ \|LEQ\|GT\|LT | $EQ\|$NEQ\|$GEQ \|$LEQ\|$GT\|$LT |
| 18: | ⟨sum⟩ → ⟨term⟩ | I |
| | \|+ ⟨term⟩ | ⟨term⟩ |
| | \|− ⟨term⟩ | ⟨term⟩ $NEG |
| | \|⟨sum⟩{+\|−} ⟨term⟩ | ⟨sum⟩⟨term⟩ {+\|−} |
| 19: | ⟨term⟩ → ⟨factor⟩ | I |
| | \|⟨term⟩{*\|/\|./. \|MODULO} ⟨factor⟩ | ⟨term⟩⟨factor⟩{*\|/\|./.\| $MODUL} |
| 20: | ⟨factor⟩ → ⟨catena⟩ | I |
| | \|⟨factor⟩**⟨catena⟩ | ⟨factor⟩⟨catena⟩** |
| 21: | ⟨catena⟩ → ⟨prim⟩ | I |
| | \|⟨catena⟩ CONCAT ⟨prim⟩ | ⟨catena⟩⟨prim⟩ $CONCA |
| 22: | ⟨prim⟩ → UNDEFINED | $UNDEF |

| *Syntactic Rule* | *Rule of Translation* |
|---|---|
| \|⟨var⟩ | ⟨var⟩ $IN |
| \|⟨label⟩ | ⟨label⟩ $IN |
| \|(⟨expr⟩) | ⟨expr⟩ |
| \|⟨block⟩ | I |
| \|⟨procdef⟩ | I |
| \|⟨referenceprim⟩ | I |
| \|⟨listprim⟩ | I |
| \|⟨numberprim⟩ | I |
| \|⟨logicalprim⟩ | I |
| \|TAIL ⟨prim⟩ | ⟨prim⟩ $TAIL |
| \|⟨var⟩ .(⟨expr-sequence⟩). | ⟨expr-sequence⟩⟨var⟩ $IN |
| \|⟨symbolprim⟩ | I |
| 23: ⟨label⟩ → ⟨name⟩ | $VARBL ⟨name⟩ |
| 24: ⟨var⟩ → ⟨name⟩ | $VARBL ⟨name⟩ |
| \|⟨var⟩ IN | ⟨var⟩ $IN |
| \|⟨var⟩ (⟨sum-sequence⟩) | ⟨var⟩⟨sum-sequence⟩) |
| 25: ⟨expr-sequence⟩ → ⟨expr⟩ | I |
| \|⟨expr-sequence⟩, ⟨expr⟩ | ⟨expr-sequence⟩⟨expr⟩ |
| 26: ⟨sum-sequence⟩ → ⟨sum⟩ | I |
| \|⟨sum-sequence⟩, ⟨sum⟩ | ⟨sum-sequence⟩) ⟨sum⟩ |
| 27: ⟨referenceprim⟩ → AT ⟨var⟩ | ⟨var⟩ |
| 28: ⟨listprim⟩ → ⟨list⟩ | I |
| \|LIST ⟨sum⟩ | ⟨sum⟩ $LIST |
| 29: ⟨list⟩ → .( ). | I |
| \|⟨listhead⟩⟨expr⟩). | I |
| 30: ⟨listhead⟩ → .( | I |
| 31: ⟨numberprim⟩ → ⟨number⟩ | $NUMBR ⟨number⟩ |
| \|REAL ⟨disj⟩ | ⟨disj⟩ $REAL |
| \|LENGTH ⟨catena⟩ | ⟨catena⟩ $LENGT |
| \|ABSOLUTE ⟨sum⟩ | ⟨sum⟩ $ABSOL |
| \|INTEGER ⟨sum⟩ | ⟨sum⟩ $INTEG |
| 32: ⟨logicalprim⟩ → TRUE | $TRUE |
| \|FALSE | $FALSE |
| \|LOGICAL ⟨sum⟩ | ⟨sum⟩ $LOGIC |
| \|⟨sypeinquiry⟩⟨var⟩ | ⟨var⟩⟨typeinquiry⟩ |
| 33: ⟨typeinquiry⟩ → ISNU | $ISNU\|$ISLO\|$ISLA |
| \|ISLO\|ISLA\|ISLI | \|$ISLI\|$ISPR\|$ISRE |
| \|ISPR\|ISRE\|ISSY\|ISUN | \|$ISSY\|$ISUN |
| 34: ⟨symbolprim⟩ → .*⟨6-symbol string⟩ | I |
| 35: ⟨procdef⟩ → ⟨prochead⟩⟨expr⟩ $. | I |
| 36: ⟨prochead⟩ .$ | .$— |
| \|⟨prochead⟩⟨formaldec⟩., | ⟨prochead⟩⟨formaldec⟩ |
| 37: ⟨formaldec⟩ → FORMAL ⟨name⟩ | $FORMA ⟨name⟩ |
| ⟨formaldec⟩, ⟨name⟩ | $FORMA ⟨name⟩⟨formaldec⟩ |
| 38: ⟨6-symbolstring⟩ | I |
| {⟨letter⟩\|⟨digit⟩⟨blank⟩ | |
| \|,\|.\|$\|*\|?\|=\|+\|− | |
| \|)\|(}^6 | |
| (i.e., a string of 6 characters.) | |
| 39: ⟨name⟩ → ⟨letter⟩ | I |

| *Syntactic Rule* | *Rule of Translation* |
|---|---|
| \|⟨name⟩⟨letter⟩ | I |
| \|⟨name⟩⟨digit⟩ | I |
| (For the IBM 7094 and the UNIVAC 1108, only the first six characters of a ⟨name⟩ are translated.) | |
| 40: ⟨number⟩ → ⟨integer⟩ | Converted to octal. |
| \|⟨integer⟩.⟨integer⟩ | Converted to octal floating point. |
| 41: ⟨integer⟩ → ⟨digit⟩ | — |
| \|⟨integer⟩⟨digit⟩ | — |
| 42: ⟨digit⟩ → 0\|1\| . . . \|9 | I |
| 43. ⟨letter⟩ → A\| . . . \|Z | I |

# SYMPLE—A general syntax directed macro preprocessor

*by* JAMES E. VANDER MEY

*The Pennsylvania State University*
University Park, Pennsylvania

ROBERT C. VARNEY

*The Pennsylvania State University*
McKeesport, Pennsylvania

and

ROBERT E. PATCHEN

*IBM Corporation*
Boston, Massachusetts

## INTRODUCTION

The subject of this paper is a general syntax directed macro preprocessor system. One of the suggested potential uses of this system is that of evaluating new or extended programming languages by the technique of syntax directed macros. This led to the association of the acronym SYMPLE (**SY**ntax **M**acro **P**reprocessor for **L**anguage **E**valuations) with this system.

A preprocessor is a processor intended to be used prior to another processing stage. In our case, it is assumed that the SYMPLE preprocessor system will generally be used in processing higher level language texts (ones which are user oriented), producing output text in the same or a similar higher level language.

The term "macro" is used in a very general sense in this paper. As in other macro systems, the macro mechanism consists of the recognition of a macro "reference" in the source text being processed, and a macro "definition" defining a translation procedure invoked by some corresponding macro reference.

A SYMPLE macro definition consists of two parts: the "macro semantic portion" or "macro body"; and the "macro templates."

> The macro semantic portion is the translation procedure and consists of the instructions to be executed when the macro is "invoked". A macro is invoked when a pattern described in one of its macro templates is recognized by the parser in the source input text. This macro reference pattern may have identifiable parts which are then considered as arguments for the semantic portion.
>
> A macro template defines a possible macro reference pattern for this macro and consists of two distinct parts: A specification of a general syntactic substructure of the source input text in which a given macro reference may occur (i.e., context); and any necessary further syntactic qualifications within that general syntactic substructure (e.g., a specific pattern). The actual pattern matching technique for macro reference is thus a two level syntax directed matching procedure. This syntax

directed macro reference technique is the method by which SYMPLE achieves both simplicity and generality.

The SYMPLE system as a macro system is not tied to any particular programming language. The base (source input) language and the object (output) language of the macro facility could in fact be entirely different languages.

The syntax of the languages to be processed and/or extended must be adequately described through the syntax description metalanguage of the SYMPLE system. This syntactic description is used for determining "context" for macro references and thus the requirements for a minimally "adequate" syntactic description of a language are proportional to the degree of context required to isolate macro references.

As a very simple example, assume all macro references must occur in only a single specific syntactic unit (syntactic substructure) of the base language (e.g., only labels of Fortran statements). Then to facilitate the recognition of macro references in the source language, the syntax of the base language need only be described via the metalanguage to the extent that it can isolate this syntactic unit type (i.e., Fortran labels.) When recognized, this syntactic unit will then be considered as a candidate for containing a macro reference.

After a candidate syntactic unit is isolated in the source input a check can be made for the existence of specific macro references by testing for further qualifying patterns within that syntactic unit. For instance, a Fortran label of "three blanks followed by two numbers" might be a specific macro reference. A check would thus be made for this reference according to the syntactic pattern defining "three blanks followed by two numbers" whenever a Fortran label is recognized. This process of local syntax investigation is called "template matching" for a macro reference.

It is also through the template matching facility that translation parameters in the source language (e.g., arguments, conditions, etc.) are recognized and passed to the actual macro facility. These translation parameters, which we shall call argument strings, can be manipulated by the instructions contained in the body of the macro (semantic portion).

Since the primary function of the SYMPLE system is that of a preprocessor, the translation process is mainly that of a manipulation of argument strings and the insertion of modified and/or created strings back into the source input. Hence, the actual semantic portion of the macro is implemented in a language oriented to the manipulation of character strings. Thus translation due to macro references and related translation parameters generally results in the insertion of the translation code in the base language into the body of the code being processed. It will be shown that this "in place" translation in the SYMPLE system does not necessarily imply expansion in exactly the same place (i.e., at the lexicographical location of the macro reference).

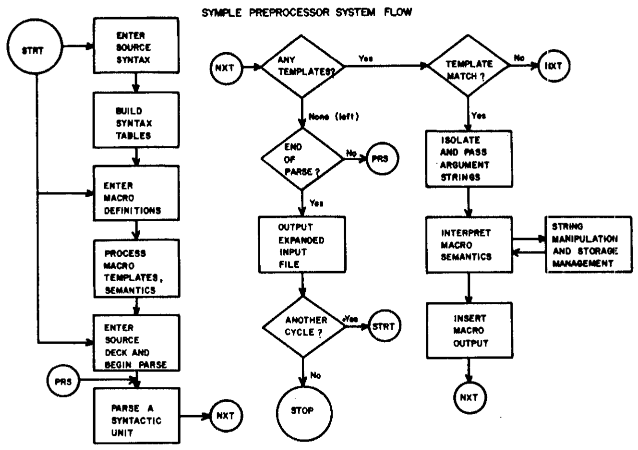

SYMPLE PREPROCESSOR SYSTEM FLOW
STRT
ENTER SOURCE SYNTAX
BUILD SYNTAX TABLES
ENTER MACRO DEFINITIONS
PROCESS MACRO TEMPLATES, SEMANTICS
ENTER SOURCE DECK AND BEGIN PARSE
PRS
PARSE A SYNTACTIC UNIT
NXT
NXT
ANY TEMPLATES?
Yes
None (left)
END OF PARSE ?
No
PRS
Yes
OUTPUT EXPANDED INPUT FILE
ANOTHER CYCLE ?
Yes
STRT
No
STOP
TEMPLATE MATCH ?
No
IXT
Yes
ISOLATE AND PASS ARGUMENT STRINGS
INTERPRET MACRO SEMANTICS
STRING MANIPULATION AND STORAGE MANAGEMENT
INSERT MACRO OUTPUT
NXT

Figure 1—A general flow of the SYMPLE macro preprocessor system

An attempt will now be made to summarize and interrelate the functions of the SYMPLE system by outlining the system functional flow via a system flow diagram (Figure 1) and the following brief description.

The preprocessor operates as follows:

1. The first items processed contain control information which includes such items as the device(s) from which subsequent information is to be read, the device(s) designed for system output, the names of special edit macros, specific listing options, etc. Control information may occur in the input stream at other logical stages of processing.
2. A description of the base language syntactic structure is read as input and processed to build a data base for the recognition portion. This data base will be used later by a parser.
3. Macros (templates and associated semantic translation routines) are read in, stored, and used to create necessary data bases for later processing.
4. A source deck is read in and parsing of the source input begins. (Probable entry point for most users.)
   a. As a syntactic unit is recognized, a check is made to see if any macros have templates to be matched in this syntactic unit.

Templates of edit macros, if any, are tested last. When there are no templates left to be checked and if the end of the total parse has not been encountered, the parse is continued.

b. If a macro template match is successful, the argument strings are passed to its associated macro semantic portion. There may be any number of macro templates associated with a given macro semantic portion, and identical template patterns can be associated with different macro semantic portions.

c. The instructions in the current macro semantic portion are executed (actually interpreted) and the results of their operations are effected (e.g., storage manipulation, insertion of translation into input source, dynamic creation of new macro templates or semantics for this or other macros). Upon completion of execution control is returned to 4a above.

5. When the source deck has been completely parsed and thus source time translations, including any necessary editing, have been completed, the file is then ready for output in a manner specified by the control information.
6. Processing is now completed, but by appropriate control information another cycle may be initiated on (a) new information or (b) on a previous preprocessor output file. Thus, in the latter case, we have the possibility of a multipass preprocessor, if desired.

The remainder of this paper will be devoted in the main to the details of what the SYMPLE system can do and in general how one goes about using the SYMPLE system. The syntax description metalanguage is introduced first followed by an introduction to the macro translation (semantic) and insertion capabilities of SYMPLE.

### *Syntax description metalanguage*

The syntax description metalanguage is used to describe a parsing "grammar" of the base language in which macro references are to be embedded and thereby outline the manner in which the source input is to be parsed. For example, suppose a label field is one syntactic structure to be parsed. The parser should then be told that a label field consists of, say, five characters which are either all digits, all blanks, or a string of blanks followed by a string of digits.

The grammatical metalanguage used to direct SYMPLE's parser is similar to the Backus-Naur Form[4] (BNF) metalanguage. For example, similar grammatical productions are used to define syntactic structures; the nonterminals and terminals of BNF are also used being renamed syntactic units and literal strings, respectively. There are, however, several features in SYMPLE's metalanguage which were incorporated to extend the power and simplicity of grammatical description over that of standard BNF.

Actual productions in SYMPLE's metalanguage to define the parsing desired in the preceding example are

```
(LABEL-FIELD):5&5(0$‘ ’0$(DIGIT))

(DIGIT):‘0’|‘1’|‘2’|‘3’|‘4’|‘5’|‘6’|‘7’|‘8’|‘9’
```

The first production above is interpreted as: a label field is defined as not less than five nor more than five characters of a string of zero or more blanks immediately followed by zero or more digits.

### Productions

The syntactic units of the base language are defined by productions in the metalanguage. These productions are of the form:

```
(LHS): right side
```

where (LHS) represents the syntactic unit being defined on the left side and the right side contains metalinguistic descriptions of other syntactic unit(s) and/or literal string(s) in the left to right order in which they comprise the structure of (LHS). The colon (:) separates the defined syntactic unit on the left side from the defining information on the right side.

The first production of the base language grammar must be the definition of the syntactic unit representing the total syntactic structure of the base language (i.e., the initial or distinguished symbol of BNF). Other productions may be in any order.

### (Named) Syntactic units

The metalinguistic representation of a syntactic unit in a production is a string of arbitrary length enclosed in parantheses. The string (called the name of the syntactic unit) may be composed of any characters with the exception of those used as special delimiters in the syntax description metalanguage (i.e., illegal characters are ():;‘|$&).

## Literal strings

A literal string is represented in the metalanguage by the desired string of characters enclosed in single quotation marks ('). Any character may be used within a literal string, except that a single quotation mark is represented by two adjacent single quotes for each occurrence in the literal string in order to differentiate it from the ending delimiter of the literal string.

## Alternatives

If a syntactic unit in the base language may have alternative representations, these alternatives may be represented in the metalanguage as a single production with the alternatives of the syntactic unit each appearing on the right side and separated from each other by the conventional OR symbol (|).

Example: (DIGIT):'1'|'2'|'3'|(OTHER)

## Complex substructures (Unnamed syntactic units)

If one does not wish to break down and label a syntax substructure in detail, but simply label an entire complex substructure as a syntactic unit, pairs of parentheses may be used as grouping indicators. Consider the following equivalent examples of a definition of the syntactic unit (NUM4).

```
Example:  (NUM):'2'|'3'|'4'
          (NUM2):'3'|'4'|'5'
          (NUM3):'5'|'6'|'7'
          (NUM4):'1' (NUM) (NUM2) |'1'
            (NUM3)
Example:  (NUM4): '1' (('2'|'3'|'4')('3'|'4'|'5')|
            ('5'|'6'|'7'))
```

Grouping may occur to any depth desired and each quantity within the grouping parentheses must have the form of any legal right side of a production.

## Quantity repetition and bounds

Often in the syntax of a base language a (named or unnamed) syntactic unit or literal string may be required to occur several times. Or it may be desirable to specify that a syntactic structure be a function of the length of an input string in addition to other qualifications (e.g., a label field of exactly five characters and consisting of . . . ).

To indicate either the repetition of a string (i.e., the input string defined by a syntactic structure) or the length bound on the number of characters in some string, an operator group must precede the respective quantity in the syntax. The operator group is of the form n$m or n&m for the string and character counters respectively, where n is an integer representing the lower bound and m, an integer representing the upper bound.

Consider the following example.

```
(A):  3$3 (SUB-STRUCTURE)
(B):  3$3 (SUB-STRUCTURE)
(C):  'C'
(SUB-STRUCTURE):    0$5  (C)
  1$3'AB'
```

The first production defines (A) as exactly three strings of (0$5(C)1$3'AB'). Thus, acceptable strings for (A) might be ABABAB or ABCABCCCCABAB or CCABABCABAB, etc. However, (B) is defined as exactly three characters which are otherwise defined as in (A). Thus, (B) can be only CAB; no other combinations will yield exactly three characters. Notice that the string counter differs from the character counter in that it is distributed over all inner strings whereas the character counter represents an absolute bound over a given substructure.

When productions include quantities with repetition counts, the parser which utilizes these productions will attempt to find the largest number of those quantities in the input source consistent with the upper bound of repetitions. If the input contains more than the upper bound of these quantities, the input string corresponding to the upper bound count of quantities will be recognized and succeeding repetitions will be analyzed according to the syntax following. A lower bound count of zero is allowable and simply indicates the optional omission of the quantity.

The absence of an explicit lower bound implies a lower bound of one. The absence of an explicit upper bound implies an upper bound which is the maximum bound allowable in the system. In the present implementation it is 32767. It should be noted that

1$1(SYUN) and (SYUN) are equivalent as are

$(SYUN) and 1$32767 (SYUN)

## Complement look-ahead

The symbol ¬ preceding a literal string, syntactic unit or grouping indicates that at that point in the syntax the quantity indicated must not occur. This is called a complement look-ahead for the indicated quantity at parse time. If the quantity is found, the parse being attempted has failed. (Any syntactic units found on the look-ahead will *not* result in macro template match attempts.) If the quantity is not found, the parse continues as before the complement look-ahead.

Example: (LETTER):'A'|'B'|'C'|'D'|'E'

(SPLTRSTRG):$(¬'C'(LETTER))

The strings recognized as (SPLTRSTRG) will be any string which consists of one or more of A, B, D or E, but not C.

**Scan positioning**

The production defining a syntactic unit can be made to include, without investigation as to structure, an arbitrary lengh of input, or it may require that a particular syntactic unit in the input conform to more than one syntactic structure. This is done by explicitly positioning the location at which the parser is "looking." This location, called the scan position, can be adjusted either relative to its present position or to the beginning reference points in the syntax of the parsed input.

a—X(Space) positioning

The occurrence of the symbol X immediately followed by an unsigned integer number and delimited by bracketing commas at any point in the right side of a production will cause the scan position to be adjusted rightward from its present location the integer number of positions specified. The symbol X and following number must be bracketed on both sides by commas except in the following cases: X is the first (last) symbol of a grouping level or the first (last) symbol of the right side of a production, in which case the left (right) comma is not required.

Example: Define an (END-CARD) to be an 80 character string. The first six characters must be blanks, the next 66 characters must have the word END somewhere with the rest blanks, and the last eight characters may be anything.

(END - CARD): 6 & 6' '66 & 66 (0$' ' ('END') 0$' '), X8

b—T (Tab) positioning

The format is similar to that of X positioning, except a T is used instead of an X.

The T scan positioning results in the scan position being moved the specificed number of places to the right of the beginning location at which the parse began at (1) this grouping level, if the T positioning appears within a grouping parenthesis pair, or (2) the right side of the production otherwise.

Example: A syntactic unit (EMPLOYEE-NO.) is defined to be an 80 character string with a syntactic unit (LAST-NAME) beginning in position one, followed by a single blank and then the syntactic unit (FIRST-NAME). Exactly 15 spaces after the beginning of (FIRST-NAME) is to appear the syntactic unit (CODE). Finally (NUMBER) will be 75 spaces from the beginning of (EMPLOYEE-NO.).

(EMPLOYEE-NO. ): (LAST-NAME) ' ' ((FIRST-NAME), T15, (CODE)), T75, (NUMBER)

*Recursive grammars in the metalanguage*

Recursive grammars (i.e., productions with the syntactic unit of the left side occurring as well on the right side, or being in the derivation of a syntactic unit of the right side) are allowed in the metalanguage subject to certain conditions.

For instance, left recursive productions are not allowable, but other recursive productions are allowable.

Further, the character (&) bound counts are cumulative from the initial (top) occurrence in a recursive parse while the repetition bounds ($) are effective at *each* level of recursion.

*Non-specific grammars in the metalanguage*

Let a non-specific grammar be one in which the particular alternatives of structure for a syntactic unit may have structurally the same *headings* (i.e., leading components which are structurally the same). The metalanguage allows the specification of such grammars and at recognition time the parser always picks the first specified (or left most) alternative as its initial guess. Subsequent guesses continue with the next specified alternatives.

The user must be aware of the possible consequences if the apparent ambiguity in a non-specific grammar causes the recognition of syntactic units to be rejected later as a result of an unsuccessful parse. Though the back-up to the next alternative is handled automatically by the parser, the syntactic units recognized may result in macro invocations; the results of which will *not* automatically be negated. Relevant user aids in this area are provided by the system.

The following example illustrates a parsing grammar for a language which is context sensitive and not context free and which utilizes recursive productions.

$$L = (0^n 1^n 0^n : n \geq 1)$$

(LANG):(LSTR) ¬ '1', T1, $'0'(RSTR)

(LSTR):'0'(LSTR)'1'|'01'

(RSTR):'1'(RSTR)'0'|'10'

The parser first determines that the input string belongs to the context-free language $0^n1^nx$; checks to make sure x does not begin with a 1; repositions to the beginning of the parsed substring of 1's and then determines that the remaining substring of the input string belongs to the context-free language $1^n0^n$. If the above conditions are true, then the input string belongs to the context-sensitive language $0^n1^n0^n$.

*The SYMPLE macro facility*

The macro facility of SYMPLE provides the actual translation mechanisms. The macros themselves are read in to the system following the base language grammar and prior to the user's source deck. The individual macro definitions are described in this section.

MACRO FORMAT

The overall format of an individual macro definitions is as follows:
<macro name> (<syntactic unit>) = <template body> / (<syntactic unit>) = <template body>....;
macro semantic statements
END;

The exact format and meaning of the various parts are described in the balance of this section.

### Macro name

The first item to appear in the macro is the name of the macro. The name may be any string of characters, excluding those special characters previously mentioned as excluded from a syntactic unit name. The macro name is used exclusively as a "handle" for the user's organization and SYMPLE's internal system and macro referencing. The macro name should not be confused with a macro reference in the source text. A source reference to the macro is completely independent of its name.

### Templates

Following the macro name are a series of macro templates which are descriptions of possible macro references that will cause the invocation of the macro. A single macro template is of the form:

(<syntactic unit>) = <template body>

where the syntactic unit is any syntactic unit that may occur in the base language, and the template body, if present, consists of a description of a specific structure to be found within that syntactic unit. The syntax and semantics of template body are identical with those of the metalanguage of SYMPLE except for an extension to make it possible to identify and name argument strings for the macro.

The extension added to facilitate the identification and naming of argument strings was simply to allow the enclosing of the desired argument location in the syntactic structure of the template within bracketing parentheses and preceding the left enclosing parenthesis with a name (with the same character restrictions as a macro name) to be associated with the enclosed argument string. These enclosed argument strings may occur anywhere within the template, and in fact may even enclose other argument strings. The names associated with the argument strings must be unique within a single macro template.

A macro template may cause a macro invocation in the following manner. When the syntactic unit designated on the left of the equal sign in a macro template is recognized by the parser, the actual structure of the syntactic unit found is compared with the specific syntax specified in the template body. A successful comparison results in the invocation of the macro and the passing to the macro of identified argument strings in the macro reference, if any. If no template body is specified, then the macro is immediately invoked with no arguments passed.

The syntax structure defined in a template body need not be structurally consistent with that of the object syntactic unit in which it will be compared. However, if the template body contains syntactic units, these units must have been in the productions submitted with the description of the base language. These productions though can be stand-alone productions (not logically in the normal base language structure) included solely for use within templates. The use of these stand-alone syntactic units, literal strings, and alternative arrangements and selection of syntactic units in the base language can result in template structures quite different from those recognized in the process of finding the object syntactic unit. Thus the template comparison is actually an attempted reparsing within the physical bounds of the object syntactic unit according to the template syntax description.

Any number of macro templates may follow the macro name, with a slash (/) separating each, except that the last template is followed by a semicolon (;).

```
Example:   NO1 (LABEL) = A1 ('    ' A2
              ((NUM)))/(STMT) = 'C' A3 (X79);
           macro semantic statments
           END;
```

Macro NO1 will be invoked when either

1. A (LABEL) s found consist'ng of four blanks followed by a (NUM), or else
2. A (STMT) is found beginning with the letter C.

In case 1 two argument strings will be available for manipulation and testing by the macro semantic statements; that associated with the string name A1 will be four blanks and the found (NUM); that associated with the string name A2 will be just the found (NUM). In case 2, the argument string associated with string name A3 will be the 79 characters following the initial letter C.

Argument string names which are not in a matched template or which are associated with null argument strings in the matched template are associated with the null string (i.e., have a length attribute of zero).

### Macro semantics

a—General

The macro semantics facility in SYMPLE is based on a string oriented language which drives an interpretive mechanism. This language closely parallels SNOBOL and has a simple syntax. The basic form of most semantic statements is

<action verb>, <string name> = <string reference>, <string reference> . . . ;

where the action verb is a key word describing some action to be performed on the referenced strings (literal strings, string names, etc.) with the resultant string generally being associated with the given string name. The details of the semantic language facility are described in another paper.[13] The use of relatively simple semantic statements in later examples should be intuitively understandable.

This semantic language provides the ability to:

1. manipulate strings of characters
2. reference strings literally, directly, indirectly
3. reference strings with concise notations
4. communicate between macros
5. execute subroutine-like macros
6. manipulate strings of values
7. alter sequential execution (branch)
8. insert strings back into the ground language code
9. loop repetitively
10. perform string comparisons
11. display string-string name associations
12. terminate interpretive action

to which needs to be added for our discussion one capability not explicitly mentioned: the ability to dynamically alter entire macros (templates and semantics).

This last capability mentioned and number 8 listed above are the means by which the macros effect their results in the translation process.

b—Output string insertion

Strings which are produced in the macro semantic portion of a macro may be inserted into the source code in any of several ways. The semantic language statement which directs the insertion of a string is of the form:

INSERT, <directive> = <string name(s)>;

The directive is a code rather than a string name which specifies the type of insertion to be performed. The directive codes are I, IA, IB, A, B, A, <digit>, B, <digit>, PI, PIA, PIB, PA, PB, PA, <digit>, PB <digit> and MADD.

They are explained below.

I—The string name(s) is an argument string name(s). The associated string is to *replace* the argument string occurrence in the macro reference.

IA—The string name(s) is an argument string name(s). The associated string is to be inserted immediately after the referenced argument string in the macro reference. In this, and for all remaining insertion directives, the macro reference itself remains unchanged.

IB—Same as IA except read "before" instead of "after".

A—The string(s) associated with the string name(s) is to be inserted immediately after the syntactic unit in which the current macro reference occurred.

B—Same as A except read "before' instead of "after."

A, <digit>—The string(s) associated with the string name(s) is to be inserted after a particular syntactic unit or grouping level of the parsed tree, called the referenced syntactic unit (RFSYUN). The RFSYUN is the first syntactic unit (at the same or higher level) to the immediate left of the syntactic unit or grouping level on the parsed tree, whose derivation includes, and is the value of <digit> levels above, the present macro reference. If a RFSYUN does not exist by the above definition then the directive A, <digit> references the beginning of the input stream.

B, <digit>—Same as A, digit except read "before" instead of "after".

P prefix directives—(e.g., PI, PIA, etc.) Each P prefix directive results in the same type of insertion as the non-prefixed directives. However, the string inserted is transparent to all future attempts at parsing or template matching (i.e., "protected"). The only exception to this is that a P prefix inserted string will be visible to the template matching of a specially designated macro, called the "edit" macro, whose name is specifiied at submission time via the processor control language. All P prefix inserted strings, if unaltered by the edit macro, will appear in their inserted locations in the final output.

c—Dynamic macro modification

In addition to inserting strings in the source submission, strings may be treated as new/changed macros via the following directive.

MADD—The string associated with the string name is a macro and includes macro templates and/or macro semantics. If the macro is new (no other macro with the same name) it will be added to the present library of macros for this submission. If the macro name is that of a current macro, macro templates, if present will be added to those presently associated with the macro and macro semantics, if present, will *replace* those of the present macro.

## CONCLUSION

The purpose of the SYMPLE system is to provide a general language-independent macro preprocessor. The syntax directed approach was used to allow both general and flexible macro referencing techniques.

The SYMPLE syntax description metalanguage was designed from the premise that the metalanguage should be a practical tool for real programming languages with their many syntactic idiosyncracies (e.g., imbedded blanks, fields of specified length, continuation columns, etc.). As far as possible and practical these real problems should be easily describable in the SYMPLE syntax description metalanguage. In a standard BNF metalanguage, such problems are at best very awkward to describe. This led to such concepts as length and repetition binding, and explicit scan positioning.

Explicit scan positioning added the abiilty to perform successive analyses, even within a local template match, by repositioning the analyzer for rescan of already parsed information. This rescanned information may of course, contain different information as the result of insertions from macro invocations.

The insertion of information in the "protected" mode (P-prefix directive insertions) further extends the power of the scan and rescan mechanism of the syntax analyzer. It allows the user the option to insert code which either may possibly affect the future syntax analysis (normal mode), or be completely "transparent" and thus not possibly affect subsequent syntactic analyses.

Systems such as TMG,[10] COGENT[14] and similar syntax directed compilers or compiler-compilers have their semantic actions hooked to the parsed syntactic units of a source submission, much the way SYMPLE would do without the local syntax parsing of a macro template. In the context of macro processors, however, the application of global syntactic analysis followed by local syntax analysis for the macro references appears to be a new application. The obvious advantage of this technique is that it provides a means of specifying a contextual dependence for macro references. Patterns in the source input which would qualify as macro references on a local syntax basis will qualify only if they are in the correct global context.

Several previous macro systems [notably XPOP,[7] ML/I,[2] LIMP[19]] use some sort of a generalized macro reference technique. Most used a template matching technique based on pseduo-syntax methods (e.g., noise word structuring of XPOP, specific literal template structures of LIMP). In each case, however, the scope of applicability of these macro references was not controlled on a global syntactic basis. ML/I, for instance, depends on the occurrence of a name of a macro in a statement for the recognition of a macro reference. XPOP looks for a macro reference in each statement based on word structures, with non-"noise" words in these structures being the arguments. Macro references in LIMP are perhaps the most general of the above mentioned systems. However, the templates of LIMP are (1) literal templates (i.e., character strings—not defined syntax structures) with "holes" in them, the "holes" being filled by the required arguments; and (2) each template is eligible in any given "line" of input. Thus there is no discrimination in regard to the applicability of a template on a global basis in any of the above mentioned systems; nor is there structuring of the templates themselves on a general syntactic basis; nor can the arguments be identified in a really general manner. It would take little to show that, at least from a macro reference point of view, these systems would be relatively simple special instances in SYMPLE.

The general applicability of the SYMPLE system has been alluded to, and a few mostly simple examples are illustrated in the appendix. These examples illustrate the use of SYMPLE as a language extension facility, in handling "sift" problems, and text editing. There are certain to be many other areas of applicability not mentioned.

## APPENDIX I

*SYMPLE processing examples*

Example 1

The first example of this appendix is designed to take OS/360 Fortran IV input and condense all non-comment statements into single condensed strings by eliminating unnecessary blanks, sequence number fields, and continuation fields. Each condensed statement will be separated by a record mark (!). Processor control information is included for completeness.

```
    SYNTAX;
    (PROG):$(STMT) (END-CARD)
    (STMT) : (LABEL-FIELD) ('0'|' ')
      (UNLAB-STMT) | (COMMENT)
    (COMMENT):'C', T81,'!'
    (LABEL,FIELD): 5 & 5((BLKSTRG) (NUM))
    (UNLAB-STMT) : ¬(END-BODY) (BLKSCN)
      (SEQFIELD) 0$19 ((CONT-FIELD)
      (BLKSON) (SEQFIELD))
    (END-CARD): 6$6 ' ' (END-BODY) (SEQ-
      FIELD)
    (END-BODY) : 66 & 66 (0$1 (BLKSTRG)
      'END' 0$1 (BLKSTRG))
    (BLKSCN) : 66 & 66 (0$1 (BLKSTRG) $
      (NONBLK))
    (BLKSTRG) : $' '
    (NONBLK):¬' '¬'"', X1|'"'$ (¬'"', X1)'"'
    (NUM):'0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
    (SEQFIELD):X7, '!'
    (CONT-FIELD):¬'C', T6, ¬('0'|' '), X1
    SYNEND;
    MACROS;
    CONDENSE (BLKSTRG) = A1 ((BLKSTRG))/
      (SEQFIELD) = A1 ((SEQFIELD))/
    (CONT-FIELD) = A1 ((CONT-FIELD));
      REPLACE, A1 = ;
    INSERT, I = A1; STOP; END;
    END-STMT (UNLAB-STMT) =; INSERT,
      A = '!'; STOP; END;
    MACEND;
SOURCE, RECMK;

          INTEEER *2   AA ( 4 )  /^1  A  B  C  D^1/^4 1
        1  BLK/^1  ^1/ , VAL ( 1 0 0 ) / 1 00*0/     2
          D O  1 0  K = 1  , 1 0 0                    3
          VAL ( K ) = VAL ( K )       +               4
          1  9 5   *   K                              5
        A  -  AA ( 4 )                                6
      1 0  AA ( K/ 1 0 0 ) =  BLL                     7
1 0 0 0 0       S T O P                               8
                             END                      9
```

Output from SYMPLE after processing above input

```
    INTEGER*2AA(4)/'A B C D'/, BLK/' '/, VAL
      (100)/100*0/!DO1OK=1,100!
    VAL(K) = VAL(K) + 95* K-AA (4)!10
    AA(K/100) = BLK! 1000 STOP! END!
```

Notes on example:

1. The grammar of Fortran IV is detailed here

only to a level which will distinguish major substructures. If one wished to further detail the syntax structure, the syntax of Fortran statements in the condensed form would be relatively simply since all extraneous clutter has been removed. The P-prefix insert capability could be used to ignore clutter for possible reparsing without actually removing it from the input (and thus output).

2. The grammar is non-specific with at least one point of apparent ambiguity. The beginning characters of an (END-CARD) will qualify as the beginning characters of a (STMT) (i.e., 6$6' '=(LABEL-FIELD)' '). Thus upon encountering an (END-CARD) there will be a back-up, since an attempt is first made to parse it as a (STMT). In this case, of course, the back-up will not have any bearing on the total processing result.
3. (LABEL-FIELD) will accept a label, say ƀƀ1ƀ5 and the compressed result would be 15. The structure of this particular (LABEL-FIELD) would be

```
     bb          1        ƀ           5
(BLKSTRG) (NUM) (BLKSTRG) (NUM).
```

4. Note how (NONBLK) includes all non-blank characters and literal strings (including blanks.)
5. The macro template (UNLAB-STMT)=; in the second macro results in the macro END-STMT being invoked with no arguments.
6. The use of the processor control RECMK parameter results in a ! being added to the end of each logical record on input. The syntax grammar used assumes this, though an equivalent grammar without the RECMK could easily be used in this case.

Example 2

This example is designed to remove all redundant parentheses in a language which uses pairs of left and right parentheses for grouping. A redundant parentheses pair is any pair of parentheses which enclos s a string which is also totally enclosed in parentheses.

```
SYNTAX;
(FLANG):$(PAREN)
(PAREN):'('(INARDS)O$1(INTOO)
(INARDS):(PAREN)|(INTOO)
(INTOO):O$(¬')'¬')',X1)O$1(PAREN)
SYNEND;
MACROS;
REDUN(PAREN)='('AA((INARDS))')';
SEPART,AA='(', AA, ')'/F, L1; INSERT, I =
  AA; L1:STOP; END;
MACEND;
SOURCE, LIST;
(((A(B)))C)((((XYZ((Q))(A))F)))
/*
```

Output from SYMPLE after processing

```
((A(B))C)((XYZ(Q)(A))F)
```

Note: In a recursive parse, inner-most (lowest) recursive syntactic units [e.g., (PAREN)] are recognized first and subject to macro expansion first.

Example 3

A final example shows a simple extension of OS/360 Fortran IV obtained by adding a different statement type to the grammar. This different statement type will contain a macro reference. The format of, and argument location in, the macro references will be strictly dependent on the local syntax specified in the templates of the macros.

A different statement type could be designated simply as starting with a non-numeric non-blank character after column 1 and before column 6. The grammar defining this basic extension could appear in a submission as follows.

```
SYNTAX,PUT;
(PROG):$(STMT) (END-CARD)
(STMT):(NEW-STMT)|(END-CARD),T80
(NEW-STMT):5&5($' '$(NONNUM-BLK)),T80
(END-CARD) : 6&6 ' ' 66&66 (0$ ' ' ('END')
  0$' '),X8
(NUM):'0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
(NONNUM-BLK):¬(NUM)¬' ',X1
SYNEND;
```

At this point the syntax description differentiatin g this new statement type is defined and any user could take advantage of the description which via the processor control PUT parameter has been saved. Using the appropriate processor control and job control statements to retrieve the above syntactic specification, a user could make submissions similar to the following.

```
SYNTAX,GET;
(NOISE) : $' '|'STORE'|'IN'|'TO'|'INTO'|'THE'|
  'PUT'|'OF'|'AND'
```

```
(NON-NOISE): $ (¬(NOISE),X1)
SYNEND;
MACROS;
SUM (NEW-STMT) = A1 ($ (NOISE) ('ADD'|
  'SUM')$(NOISE)A2((NON-NOISE)) $
  (NOISE) A3 ((NON-NOISE)) $ (NOISE)
  A4((NON-NOISE)),T80);
CONCAT, A1 = '      ', A4, '=', A2, '+', A3;
  INSERT,I=A1;
STOP;END
MACEND;
SOURCE;
C THIS IS A FORTRAN COMMENT
  ADD A TO B AND STORE IN C
        SUM A AND B AND PUT INTO C
  STORE THE SUM OF A AND B IN C
          END
/*
```

Output of SYMPLE after processing

```
C THIS IS A FORTRAN COMMENT
  C = A + B
  C = A + B
  C = A + B
        END
```

The macro used above is a simple macro using a keyword and non-noise positional parameters. The illustrated new type of statement if imbedded in any Fortran source deck, would, when processed, be converted to the Fortran type statements listed, and replace the new statements.

## REFERENCES

1 M J BAILEY M P BARNETT P B BURLESON
*Symbol manipulation in Fortran - SASP I subroutines*
CACM Vol 7 No 6 June 1964 339-346

2 P J BROWN
*The ML/I macro preprocessor*
CACM Vol 10 No 10 Oct 1967 618-623

3 J A FELDMAN
*A formal semantics for computer languages and its application in a compiler-compiler*
CACM Vol 9 No 1 Jan 1966 3-9

4 J A FELDMAN D GRIES
*Translator writing systems*
CACM Vol 11 No 2 Feb 1968 77-113

5 D E FERGUSON
*Evolution of the meta-assembly program*
CACM Vol 9 No 3 March 1966 190-196

6 M L GRAHAM P Z INGERMAN
*A universal assembly mapping language*
Proc ACM Aug 1965 409-421

7 M I HALPERN
*XPOP: A metalanguage without metaphysics*
Proc FJCC Vol 26 1964 57-68

8 M I HALPERN
*Toward a general processor for programming languages*
CACM Vol 11 No 1 Jan 1968 15-26

9 B M LEAVENWORTH
*Syntax macros and extended translation*
CACM Vol 9 No 11 Nov 1966 790-793

10 R M McCLURE
*TMG-A syntax directed compiler*
Proc ACM Aug 1965 262-274

11 M D McILROY
*Macro instruction extensions of compiler languages*
CACM Vol 3 No 4 April 1960 214-220

12 C N MOOERS L P DEUTSCH
*TRAC-A text handling language*
Proc ACM Aug 1965 229-246

13 R E PATCHEN
*String oriented macro language and interpreter*
Penn State Univ Dec 1968 Thesis in Computer Science

14 J E REYNOLDS
*An introduction to the COGENT programming system*
Proc ACM Aug 1965 422-437

15 S ROSEN
*A compiler building system developed by Brooker and Morris*
CACM Vol 7 No 7 July 1964 403-414

16 E F STORM
*CHAMP - Character manipulation procedures*
CACM Vol 11 No 8 Aug 1968 561-566

17 J E VANDER MEY
*A general syntax diretced macro preprocessor*
Penn State Univ March 1969 Thesis in Computer Science

18 R C VARNEY
*The central portion of the SYMPLE system—Tree construction and parsing*
Penn State Univ June 1969 Thesis in Computer Science

19 W M WAITE
*A language independent macro processor*
CACM Vol 10 No 7 July 1967 433-441

# An algebraic extension to LISP

*by* PRENTISS HADLEY KNOWLTON

*Harvard University*
Cambridge, Massachusetts

## INTRODUCTION

An algebraic facility for LISP is quite desirable. Such a capability is motivated by the desire to utilize the primitive LISP arithmetic functions at the algebraic expression level. The requirement for a means of evaluating expressions might very well arise from applications in algebraic manipulation. Thus, the user, having performed some sort of transformation on an algebraic expression, might wish to have the resulting expression evaluated for a specific set of values. This facility, in response to this requirement, has the acronym "LEAF" (LISP Extended Algebraic Facility).

Design considerations and FORTRAN language facilities provided by LEAF include:

1. a list structured organization compatible with existing LISP;
2. an arithmetic assignment statement;
3. a DO statement;
4. a logical IF statement;
5. an unconditional GO TO statement; and
6. an INPUT and OUTPUT statement.

Since LEAF is designed in the "spirit" of LISP, built in functions in a given LISP system which provide for such conveniences as "pretty printing" of functions and editing facilities may also be applied to LEAF programs.

### *The list structured organization of LEAF*

Although the initial motivation in developing LEAF was to extend the LISP language, a number of other motivating properties of the LEAF concept make themselves apparent as one uses the LEAF facility.

In order to attain compatibility with the existing LISP language, LEAF is essentially a dialect of FORTRAN *in list structure*. Hence, a program is a list whose elements are statements. A simple LEAF program to accept two numbers from the teletype, determine their sum, and type out the result might be written as follows:

$$\begin{array}{l}( (INPUT\ A\ B) \\ (C = A + B) \\ (OUTPUT\ \ C) )\end{array}$$

In similar manner, a statement is a list whose elements are the components of that statement. In order to execute a statement, the LEAF interpreter typically looks at the keyword (e.g., INPUT), the first element of the statement, to determine how the statement should be processed. This is analogous to the LISP interpreter, in which the first element of a LISP command is a function, and the remaining elements of that command constitute the arguments of the function.

In the "assignment" function, unlike the other LEAF commands, the keyword or "=" is the second element of the list. If the item on the left hand side of the equal sign is an array reference, the subscripting can be thought of as a single list element, a sublist whose elements constitute the subscripts. In SDS 940 LISP as well as in other LISP implementations, commas are perfectly acceptable list element delimiters. Thus, the user is free to use commas for readability in subscript lists if he desires, and he is not constrained to always delimit list elements with blanks. It is important to note in the case of a subscripted variable on the left hand side of the equal sign in the assignment statement that the "=" is in fact the third element of the list. Nevertheless, recognition and processing of the assignment statement is still a relatively straightforward procedure.

In addition to the properties LISP and LEAF share, it is interesting to note that the conveniences which exist for displaying and modifying LISP functions are also applicable to the display and modification of LEAF programs. The nesting of DO loops is readily apparent from the indented listing one obtains from the LISP "pretty printing" facility:

```
( (DO I = 1 TO 10
     (A(I) = B(I))
     (DO J = 1 TO 10
        ( . )
        ( . )
        ( . ) )
     ( . )
     ( . )
     ( . ) ) )
```

In like manner, one may utilize the editing facilities available on a given LISP system to modify a LEAF program with equivalent flexibility as modifying a LISP function.

*Justifications for a list structure*

It is worthwhile noting that the list structured approach to the design of an algebraic language lends itself well to the concepts of program block structure, program editing, adaptability to a time sharing environment, and, most important of all, language and data structure compatibility.

Program block structure of the LEAF system is best illustrated by the DO statement, in which a list whose elements are statements constitute the range of the DO specification. This program block structure lends itself well to editing operations, since, armed with an indented listing of his program, one is able to quickly and accurately access and work with his program at any level. An example of program modification using the editing facility of SDS 940 BBN LISP is given in Appendix C.

Like the LISP language, LEAF lends itself well to a time sharing environment, in that LEAF programs are easily interpreted at the source language level. List structured organization of LEAF programs permit several users to work independently with the same reentrant interpreter, even when two separate programs are "intertwined" in the same storage region.

A particularly significant observation one might make of the LEAF language is that it possesses the same basic structure as its data. Hence, there is no reason why one might not wish to devise a program which performs operations upon itself, such as the changing of a "+" to an "*" in an arithmetic expression. In this sense, within the framework of the LEAF language, a statement might be thought of as an alphanumeric vector whose elements are keywords, operators, and operands.

*Fortran language facilities provided by LEAF*

1. The Assignment Statement

   The assignment statement of LEAF is identical to that of FORTRAN IV with the additional flexibility of mixed mode arithmetic. Thus, one may work interchangeably with both integer and real data in arithmetic expressions without worrying about problems of mode conversion, since the existing LISP floating point functions are designed to handle such situations automatically.

2. The DO Statement

   The DO specification of LEAF is similar to that of PL/I. The remainder of the statement consists of a list whose elements as statements constitute the range of the DO. Any level of nesting is permissible, and the LISP "pretty printing" facility shows the nesting quite clearly as illustrated earlier.

3. The Logical IF Statement

   Like PL/I, the logical IF statement consists of an "IF" part followed by a "THEN" part. The "IF" part consists of two arithmetic expressions separated by a relational operator (without periods). The true or false value of the relation determines the execution or non-execution of the "THEN" part. In either event, the next statement in sequence is reached.

4. The Unconditional GO TO Statement

   The GO TO statement of LEAF, like that of PL/I, specifies destination by means of a name rather than by means of a statement number as is the case with FORTRAN IV.

5. The INPUT Statement

   The INPUT statement consists of the key word "INPUT" followed by the variables to be defined. The "RATOM" (read atom) function of SDS 940 BBN LISP permits relative free formatting of input data.

6. The OUTPUT Statement

   Similarly, the OUTPUT statement consists of the keyword "OUTPUT" followed by the variables to be printed. The "PRINT" function of SDS 940 BBN LISP is utilized in this context.

## CONCLUSIONS

The LEAF approach seems to be an answer to certain problems facing users who are dissatisfied with present day LISP and present day FORTRAN. Feasibly, programs already written in FORTRAN IV might be converted to LEAF. The advantages of indented display of program nesting as well as the facilities of the LISP editor would certainly warrant this activity.

Working with an algebraic language at the source language level has many distinct advantages. Among these advantages, this writer suggests that the COMMENT statement should be treated as an executable statement, whose text could be made to be listed by user request during program execution.

The author sincerely hopes that the philosophy of the LEAF system is given some consideration by the implementers of future algebraic compilers.

## ACKNOWLEDGMENTS

The author wishes to extend special thanks to Dr. Daniel G. Bobrow of Harvard University's Applied Mathematics Department, under whom this work was done as independent study. Dr. Bobrow is also responsible for many of the facilities present in SDS 940 BBN LISP.[1] Special thanks are also due to Aiken Computation Laboratory of Harvard University who graciously provided SDS 940 computer time for the carrying out of this work.

Mr. Cornelius Peterson, manager of the Boston Office of Computer Usage Company, provided the necessary facilities for the writing of this paper. Finally, Mr. Burton Bloom, Senior Staff Analyst of the CUC Boston Office, provided many helpful suggestions during the technical revision of this work.

Finally, the author extends appreciation to Jet Propulsion Laboratory, Pasadena, California, for the use of their facilities in preparing visual aids in the presentation of this paper.

## REFERENCE

1 D G BOBROW et al
*The BBN 940 LISP system*
Bolt Beranek and Newman Inc Cambridge Mass April 1968

## APPENDIX A

*Syntax description of the LEAF system*

*I. Fundamental Language Components:*

*⟨letter⟩* :: = *A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z*
*⟨digit⟩* :: = *0|1|2|3|4|5|6|7|8|9*
*⟨identifier⟩* :: = *⟨letter⟩* { *⟨letter⟩* | *⟨digit⟩*}$_0^\infty$
*⟨variable⟩* :: = *⟨identifier⟩*
*⟨unsigned-integer-constant⟩* :: = *⟨digit⟩* { *⟨digit⟩*}$_0^\infty$
*⟨sign⟩* :: = + | —
*⟨integer-constant⟩* :: = [*⟨sign⟩*] *⟨unsigned-integer-constant⟩*
*⟨real-constant⟩* :: = [*⟨sign⟩*] *⟨unsigned-integer-constant⟩*.
| [*⟨unsigned-integer-constant⟩*] [*⟨exponent-part⟩*]|
[*⟨sign⟩*] [*⟨unsigned-integer-constant⟩*].
*⟨unsigned-integer-constant⟩* [*⟨exponent-part⟩*]|
[*⟨sign⟩*] *⟨unsigned-integer-constant⟩* *⟨exponent-part⟩*
*⟨exponent-part⟩* :: = [*⟨sign⟩*] { *⟨digit⟩*}$_1^2$

*II. Basic Language Elements*

*⟨program⟩* :: = ({ *⟨statement⟩*}$_1^\infty$)
*⟨statement⟩* :: = (*⟨comment-statement⟩*)|(*⟨optional-statement-label⟩* *⟨statement-body⟩*)
*⟨comment-statement⟩* :: = *COMMENT* *⟨commentary⟩*|**⟨commentary⟩**

```
⟨optional-statement-label⟩ :: = [⟨identifier⟩]
⟨statement-body⟩ :: = ⟨do-statement⟩|⟨input-statement⟩|
          ⟨output-statement⟩|⟨assignment-statement⟩|
          ⟨go-to-statement⟩|⟨if-statement⟩|⟨stop-statement⟩
⟨do-statement⟩ :: =DO ⟨index⟩ = ⟨initial-value⟩ TO ⟨final-value⟩
          (⟨do-block⟩)
⟨do-block⟩ :: = {⟨statement⟩}$_1^\infty$
⟨input-statement⟩ :: = INPUT ⟨argument-list⟩
⟨argument-list⟩ :: = {⟨variable⟩}$_1^\infty$
⟨output-statement⟩ :: = OUTPUT ⟨argument-list⟩
⟨assignment-statement⟩ :: = ⟨variable⟩ = ⟨arithmetic-expression⟩
⟨arithmetic-expression⟩ :: = ⟨term⟩ ⟨plus-or-minus⟩ ⟨arithmetic-expression⟩|⟨term⟩
⟨plus-or-minus⟩ :: = + | −
⟨term⟩ :: = ⟨factor⟩ ⟨star-or-slash⟩ ⟨term⟩ | ⟨factor⟩
⟨star-or-slash⟩ :: = * | /
⟨factor⟩ :: = ⟨variable⟩ | ⟨constant⟩ | (⟨arithmetic-expression⟩)
⟨constant⟩ :: = ⟨integer-constant⟩ | ⟨real-constant⟩
⟨go-to-statement⟩ :: = GO TO ⟨identifier⟩
⟨if-statement⟩ :: = IF ⟨arithmetic-expression⟩ ⟨relational-operator⟩
          ⟨arithmetic-expression⟩ THEN (⟨statement⟩)
⟨relational-operator⟩ :: = GT|GE|LT|LE|EQ|NE
⟨stop-statement⟩ :: = STOP
```

## APPENDIX B

*Some representative functions of the LEAF interpreter*

```
(STATEMENT
  (LAMBDA (COMMAND)
    (COND
      ((COMMENT-STATEMENT COMMAND)
        NIL)
      ((DO-STATEMENT COMMAND)
        NIL)
      ((INPUT-STATEMENT COMMAND)
        NIL)
      ((OUTPUT-STATEMENT COMMAND)
        NIL)
      ((ASSIGNMENT-STATEMENT COMMAND)
        NIL)
      ((GO-TO-STATEMENT COMMAND)
        NIL)
      (T (IF-STATEMENT COMMAND)))))

(COMMENT-STATEMENT
  (LAMBDA (COMMAND)
    (EQ (CAR COMMAND)
      (QUOTE COMMENT))))

(DO-STATEMENT
  (LAMBDA (COMMAND)
  (PROG (INDEX FROM TO)
        (COND
          ((NEQ (CAR COMMAND)
                (QUOTE DO))
            (RETURN NIL)))
        (SETQ INDEX (CADR COMMAND))
        (SETQ FROM (CADDDR COMMAND))
        (SETQ INDEX FROM)
        (SETQ TO (CADDDDDR COMMAND))
  LOOP (COND
          ((GREATERP INDEX TO)
            (RETURN T)))
        (LEAF (CADDDDDDR COMMAND))
        (ADD1 INDEX)
        (GO LOOP)
     )))

(INPUT-STATEMENT
  (LAMBDA (COMMAND)
    (PROG (ARGUMENT-LIST)
        (COND
          ((NEQ (CAR COMMAND)
                (QUOTE INPUT))
            (RETURN NIL)))
        (SETQ ARGUMENT-LIST (CDR COMMAND))
    LOOP (COND
          ((NULL (CAR ARGUMENT-LIST))
            (RETURN T)))
        (SET (CAR ARGUMENT-LIST)
          (RATOM NIL))
        (SETQ ARGUMENT-LIST (CDR ARGUMENT-LIST))
        (GO LOOP)
     )))

(OUTPUT-STATEMENT
  (LAMBDA (COMMAND)
    (PROG (ARGUMENT-LIST)
        (COND
          ((NEQ (CAR COMMAND)
                (QUOTE OUTPUT))
            (RETURN NIL)))
        (SETQ ARGUMENT-LIST (CDR COMMAND))
    LOOP (COND
          ((NULL (CAR ARGUMENT-LIST))
            RETURN T)))
        (PRINT (CAAR ARGUMENT-LIST))
        (SETQ ARGUMENT-LIST (CDR ARGUMENT-LIST))
        (GO LOOP)
     )))

(ASSIFNMENT-STATEMENT
  (LAMBDA (COMMAND)
   (PROG NIL
         (COND
           ((NEQ (CADR COMMAND)
               (QUOTE =))
             (RETURN NIL)))
         (SET (CAR COMMAND)
           (ARITHMETIC-EXPRESSION (CDDR COMMAND)))
         (RETURN T)
     )))

(ARITHMETIC-EXPRESSION
  (LAMBDA (LIST)
    (PROG (VALUE)
          (SETQ POINTER LIST)
          (SETQ VALUE (TERM NIL))
      LOOP (COND
            ((NULL (CAR POINTER))
              (RETURN VALUE))
            ((EQ (CAR POINTER)
                (QUOTE +))
              (SETQ POINTER (CDR POINTER))
              (SETQ VALUE (FPLUS VALUE (TERM NIL)))
              (GO LOOP))
            ((EQ (CAR POINTER)
                (QUOTE —))
              (SETQ POINTER (CDR POINTER))
              (SETQ VALUE (FDIFFERENCE VALUE (TERM NIL)))
                (GO LOOP))
            (T (RETURN VALUE)))
      )))

(TERM
  (LAMBDA NIL
    (PROG (VALUE)
            (SETQ VALUE (FACTOR NIL))
      LOOP (COND
              ((NULL (CAR POINTER))
                (RETURN VALUE))
              ((EQ (CAR POINTER)
                  (QUOTE *))
                (SETQ POINTER (CDR POINTER)
                (SETQ VALUE (FTIMES VALUE (FACTOR NIL)))
                  (GO LOOP))
              ((EQ (CAR POINTER)
                  (QUOTE /))
                (SETQ POINTER (CDR POINTER))
                (SETQ VALUE (FQUOTIENT VALUE (FACTOR NIL)))
                  (GO LOOP))
          (T (RETURN VALUE)))
      )))

(FACTOR
 (LAMBDA NIL
   (PROG (VALUE POINTER-SAVE)
         COND
           ((NUMBERP (CAR POINTER))
             (SETQ VALUE (CAR POINTER))
             (SETQ POINTER (CDR POINTER))
             (RETURN VALUE)
           ((ATOM (CAR POINTER))
             (SETQ VALUE (CAAR POINTER))
             (SETQ POINTER (CDR POINTER))
             (RETURN VALUE)
           (T (SETQ POINTER-SAVE POINTER)
             (SETQ VALUE (ARITHMETIC-EXPRESSION (CAR POINTER)))
             (SETQ POINTER POINTER-SAVE)
             (SEQ POINTER (CDR POINTER))
             (RETURN VALUE)))
      )))

(FDIFFERENCE
  (LAMBDA (A B)
    (FPLUS A (FMINUS B))))
(LEAF
  (LAMBDA (PROGRAM)
    (PROG (LOCATION LABEL
           (SETQ LOCATION PROGRAM)
      LOOP (COND
           ((NULL (CAAR LOCATION))
             NIL)
           ((STOP-STATEMENT (CAR LOCATION))
             (RETURN (QUOTE STOP))))
         (STATEMENT (CAR LOCATION))
         (SETQ LOCATION (CDR LOCATION)
         GO LOOP)
      )))

(STOP-STATEMENT
  (LAMBDA (COMMAND)
    (EQ (CAR COMMAND)
      (QUOTE STOP))))
```

## APPENDIX C

*Representative applications of the LEAF system*
*Examples of input statements, output statements, the assignment statement, and arithmetic expressions*

```
← INPUT-STATEMENT ((INPUT A B C D E F G))
1.0 2.0 3.0 4.0 5.0 6.0 7.0
T‡
```

‡ The "T" indicates that the invoked function succeeded.

```
← OUTPUT-STATEMENT ((OUTPUT A B C D E F G))
1.000000000
2.000000000
3.000000000
4.000000000
5.000000000
6.000000000
7.000000000
T
← ASSIGNMENT-STATEMENT ((H = A + B + C + D + E + F + G))
T
← OUTPUT-STATEMENT ((OUTPUT H))
28.00000000
T
← ARITHMETIC-EXPRESSION ((A * B * C * D * E * F * G))
5040.000000
← ARITHMETIC-EXPRESSION ((A + B * C))
7.000000000
← ARITHMETIC-EXPRESSION ((A * B + C))
5.000000000
ARITHMETIC-EXPRESSION (( ((((((((((A)))))))))) - ((B))/(C + D)
*(E + F]†
−2.142857143
← ARITHMETIC-EXPRESSION(( (A + B) * (C + D − F) ))
3.000000000
← ARITHMETIC-EXPRESSION(( A − B + C) / (D + F * (((G))) ))
4.347826087E-02
← ARITHMETIC-EXPRESSION(( A / B − C / D + F * G)
41.75000000
```

*A program using input, output, and assignment statements*

```
← E(SETQQ PROGRAM ((INPUT A B) (C = A + B) (D = A − B) (E = A * B)
                  (F = A / B) (OUTPUT A B C D E F) (STOP)) )‡
← E(LEAF PROGRAM)§
2.0 3.0
2.000000000
3.000000000
5.000000000
—1.000000000
6.000000000
6.666666667E–01
```

† The "]" causes a sufficient number of right parentheses to be generated.

‡ At this point, the atom "PROGRAM" is bound with the LEAF program as shown. The top-level function "E" merely means "execute the given function (first elements) on its arguments without prior evaluation of those arguments."

§ The LEAF interpreter is now applied to the designated program. The user satisfies the INPUT statement by typing "2.0 3.0 (CR)," and the LEAF system responds with the desired output, followed by "STOP" as generated by the STOP statement.

```
STOP
```

*A program using the DO statement*

```
← PRETTYPRINT(SUMMATION) ȣ
((SUM = 0.000000000)
     (COUNT =0.000000000)
     (DO I = 1 TO 10 ((COUNT = COUNT + 1.000000000)
          (SUM = SUM + COUNT)
          (OUTPUT SUM)))
     (STOP))

← E(LEAF SUMMATION)
1.000000000
3.000000000
6.000000000
10.00000000
15.00000000
21.00000000
28.00000000
36.00000000
45.00000000
55.00000000
STOP
```

*Modification of a program using the editing facility*

```
← EDITV(SUMMATION)‡
EDIT
*(1 (SUM = 1.0))
*3
*7
*2
*P
(SUM = SUM + COUNT)
*(4 *)
*↑
*PP
```

ȣ In this instance, we assume that the "SUMMATION" program has already been defined; hence, we need only print it out using the "PRETTYPRINT" of SDS 940 BBN LISP. Note how transparent program block structure becomes via this facility.

‡ At this point we wish to edit our sample SUMMATION example to no longer produce successive sums, but to produce successive products or factorials. The "*" tells us we are talking to the editor. The command "*(1 (SUM = 1.0))" updates the first statement of our original summation program (1.0 is the identity element for multiplication.). "*3" focuses our attention on the DO statement, "*7" focuses our attention on the range of the DO, and "*2" focuses our attention on the second statement of the range of the DO. "*P" causes that statement to be printed out, the operation "(4 *)" causes the "+" of that statement to be changed to an "*", "↑" returns our attention to the top level, "*PP" "pretty prints" the edited function, and "OK" tells the editor we are all done.

```
  ((SUM = 1.000000000)
      (COUNT = 0.000000000)
      (DO I = 1 TO 10 ((COUNT = COUNT + 1.000000000)
          (SUM = SUM * COUNT)
          (OUTPUT SUM)))
      (STOP))
*OK
SUMMATION

← E(LEAF SUMMATION)
1.000000000
2.000000000
6.000000000
24.00000000
120.0000000
720.0000000
5040.000000
40320.00000
362880.0000
3628799.999
STOP
```

# An on-line machine language debugger for OS/360

*by* WILLIAM H. JOSEPHS

*The Rand Corporation*
Santa Monica, California

## INTRODUCTION

The environment provided by the multiprogrammed options of Operating System 360 is not the most suitable for debugging. It is primarily a batch system, with a programmer's card deck disappearing into the card reader and reappearing at some future time on a printer. What happens in between is often impossible to discern; any attempt to monitor a program's execution (e.g., the setting of an address stop) is so complicated that it is nearly impossible. In this environment, debugging is difficult—at the conclusion of a program, the programmer either has successful execution or some indication of program error. If he planned ahead (and was lucky), his output will include not only an indication of the actual error, if one occurred, but trace information (either through OS TESTRAN facilities or his own printouts) to help him determine the problem. However, he is usually presented with a dump, containing a numerical reference to the completion-codes manual. More importantly, the dump represents the state of the system when OS decided it could not continue the program's execution; the user must discover why it went wrong by educated guesses and by "playing computer" with his program. The difficulty and sheer wastefulness of this procedure is extremely evident. For this purpose, an on-line symbolic debugger can be invaluable.

One traditional environmental requirement for on-line debugging is an on-line system with remote job-entry capabilities and file-management functions, or a dedicated machine and its operator console. DYDE (Dynamic Debugger), the system described herein, was developed in and for the former environment using the RAND Simultaneous Graphics System. However, the debugger can be used in a normal OS batch environment using any available 2260 graphic-display terminal or even the on-line operator's typewriter.

The text that follows includes an external description, including invocation procedures and command formats, followed by a brief explanation of the internal operation of the debugger (including the "pingpong" SVC).

## *DYDE*

### Invocation of DYDE

DYDE is executed as an OS job using a standard set of Job Control Statements (see Figure 1). These define the library in which DYDE resides (JOBLIB), a library containing the program or programs to be debugged (SYSLIB), and a scratch file for organizing the symbol table (SYSUT1). In addition, any JCL statements defining data sets that are used by the program to be debugged must be included (in this context, DYDE contains a facility for overriding both the SYSLIB and the SYSUT1 ddnames if the program being debugged needs them). Figure 2 illustrates a procedure for assemblying, link editing, and debugging. In any of these procedures, as soon as DYDE receives control, it writes out a message indicating its readiness for user commands.

### Device dependencies

DYDE can interact with the user through either

```
//      JOB
//JOBLIB  DD  library definition
//S1  EXEC  PGM=DYDE
//SYSLIB  DD  library definition
//SYSUT1  DD  UNIT=SYSDA,SPACE=(TRK,(5,1))
//SCOPE  DD  UNIT=040
```

Figure 1—Sample JCL for invocation of DYDE using the 2260 version

```
//     JOB
//JOBLIB  DD  library definition
//STEP1  EXEC  ASMFCL,PARM.ASM='TEST',PARM.LKED='TEST'
//ASM.SYSIN  DD  *
    source deck
/*
//STEP3  EXEC  PGM=DYDE
//SYSLIB  DD  DSNAME=*.STEP1.LKED.SYSLMOD,DISP=(OLD,DELETE)
//SAMPLEDD  DD  data set description
//SYSUT2  DD  UNIT=SYSDA,SPACE=(TRK,(5,1))
//SCOPE  DD  UNIT=040
```

Figure 2—Sample assemble, link edit, and debug JCL
Note: The SYSLIB card points to the output of the Link Edit step, and the user will override (using the *DDNAME command) the SYSUT1 default name with SYSUT2

an IBM 2260 display station or the IBM 1052 operator's console. For this purpose, two versions of DYDE exist; one for the 2260 interaction, the other for the 1052 (described in Appendix A). Because these devices are extremely different, the mechanics of the interaction differ significantly. However, the basic operations are the same.

The more natural mode of operation, and the one for which DYDE was originally designed, uses the IBM 2260 graphics-display station. This is an alphanumeric device with a CRT capable of displaying up to twelve lines of text; each line can contain a maximum of 80 characters. The control unit for the 2260, the IBM 2848, buffers typed messages, displays typed characters, and handles display regeneration and cursor advancement. The main CPU is presented with an attention interrupt only when the enter key is depressed. The OS Graphics Access Method (GAM) schedules an asynchronous routine of DYDE that, in turn, activates the main routine in DYDE. The message is then read and acted upon.

The twelve-line screen face is divided into two logical sections:

1. The first three lines—0, 1, and 2—are for DYDE-user communication;
2. The remaining nine lines—3 through 11—are for data display.

Data is written in the second area in a wrap-around fashion—the first data item is displayed starting on line 3, the next on line 4, and so on until the screen is full. At this point, new data is displayed starting again on line 3 (erasing automatically the previously displayed data); and line 4 is erased, providing a visual delimiter between old data and the most recent display. Each new line of data display is handled in this manner, with the data overwriting the oldest data on the screen, and the next numbered line blanked as a delimiter.

The three remaining lines—0, 1, and 2—are used for command processing. The user enters his commands on line 2 beginning with a start symbol (displayed as ▶ and usually written automatically by DYDE) followed by the command; this is followed by the attention or the enter key (displayed as ■ ) that interrupts the CPU. DYDE reads the message and immediately echos (i.e., rewrites) it on line 0. This provides not only positive verification of the transmission but also, as the user prepares to type the next message, a useful indicator of the last operation performed. Any data display requested is displayed on the first free line of the data area, and the line following is blanked. Finally, DYDE writes a confirmation message on line 1 and prepares line 2 for the next command by erasing it, writing the start symbol, and positioning the cursor at the first free space. Should the command be syntactically incorrect, an error message is written on line 1—the echo message on line 0 provides the user with ready reference for discovering his error–and the data region of the display is not disturbed.

The discussion that follows is concerned primarily with the 2260 version of DYDE rather than the 1052. Significant differences will be noted; however, all command and message formats, as well as operational details, are described for the graphic station version rather than for the typewriter version.

## Typical debugging session

A typical debugging session begins when DYDE gains control and writes its READY message. At this point, the user can identify the program to be debugged, perhaps overridding one or more of the ddnames that DYDE normally uses. After the program has been successfully LOADed, the full spectrum of DYDE commands is available to the user. He may indicate to DYDE that he wishes execution of his program to be temporarily suspended when control reaches specified locations; this is done by inserting breakpoints at these locations. Commands exist for modifying parts of his code or his data. He can then request DYDE to begin execution of his program. At this point, four events can suspend program execution and transfer control to DYDE:

1. Control reaching a previously defined breakpoint;
2. Executing the pingpong supervisor call as an assembled instruction in the user's program (e.g., useful when debugging an overlay program when a particular load is not originally in core);
3. An asynchronous interrupt from the user at his 2260 (not available for 1052 users);
4. The program program checks (e.g., it specifies an invalid address or operation code).

For release 17 of the operating system, a fifth event can suspend program execution:

5. Whenever the user's program is terminated abnormally by the operating system.*

At any of the above halting points, the user may, for example: (1) display data in his program, (2) modify data, instructions, or register contents, (3) create hardcopy of specified areas within his program, (4) insert new breakpoints, or (5) delete old breakpoints. He may resume execution of his program from the point at which it last halted (the "current" breakpoint) in either the instruction step mode (execute one instruction at a time) or in the uncontrolled mode, in which case only one of the above events can suspend program execution again. In this manner, the user can watch his program's execution to catch an error as it is occurring as well as test his program with sample data or temporary patches.

## DYDE commands

The available commands that the user may issue fall into two general categories: (1) those that create the proper environment for debugging the program, and (2) those that cause actual data display from the program.

* Items two through five are considered by DYDE to be implicit breakpoints.

All "environmental" commands begin with an asterisk, followed by the command keyword. If parameters are necessary, the keyword is followed by an equal sign; then the parameters are entered and delimited by one of several special characters (the selection of the special characters is made by the user). These special characters include the following symbols: ' ', '-', ':', ';', '/', '.', '$', and '@'. In the commands descriptions that follow, the '/' is used. Most of the commands allow different forms of the parameters; however, each legal form is stated explicitly, and no other form may be used. Within the parameter descriptions, the user substitutes the indicated quantity for lower-case items and supplies the operand exactly as shown for upper-case items. Several commands contain a quantity called "loc" as a parameter. In general, this refers to a location within the user's program; its actual use is described at the end of this section.

The commands (with the preceding start symbol and the trailing, end-of-message symbol omitted) ollow.

1. *NAME=pgmname

defines the linkage-editor-assigned member name of the program to be debugged. This program is LOADed from the data set defined by the SYSLIB DD card (or any overrides—see *DDNAMES command below). While LOADing the program, the debugger organizes the symbol table, if present, and writes it out on the data set defined by the SYSUT1 DD card (also overridable—see the *DDNAMES command). The command may be issued at any time; if a previous program is in core, it is deleted, and the debugger reinitializes itself before LOADing the new program.

2. *FINISH

terminates the debugger.

3. *PARM=parameter information

sets up pointers so that the information following the equal sign is passed to the program according to normal OS standards.*

* If the parameters are coded PARM='XYZ' on the EXEC card, the command should be *PARM=XYZ.

4. *DDNAME=syslib/sysut1/sysprint

causes the debugger to override, in its DCBs, the default name for the library data set, the symbol-table, the utility-work data set, and the data set to contain hardcopy output. The normal names are SYSLIB, SYSUT1, and SYSPRINT. However, as indicated previously, the user may need these names for his program's execution. In this case, he may, using this command, override one, all, or any combination of these three names; e.g., if the user included a DD card name PRIVLIB instead of the SYSLIB card, he would issue *DDNAME =PRIVLIB. If he needed the name SYSUT1 and SYSPRINT for his program's execution, he could include DD cards named A and B and issue the command *DDNAME=/A/B. To be effective, this command must be issued before the associated data set is needed; to issue a *NAME command followed by the *DDNAME would be meaningless unless the user wished to debug two programs from two different libraries.

5. (1) *SETMODE=NEXT=ON

(2) *SETMODE=NEXT=OFF

causes the debugger to change its global mode setting. NEXT=ON tells the debugger to recognize the next *GO (or a null command) as a command to execute the next instruction; in this way, the background program can be run one instruction at atime. NEXT=OFF resets this.

6. *TRACE

causes DYDE to print the current contents of the screen face into the SYSPRINT data set and, thereafter, to print each displayed line. If DYDE is tracing currently, *TRACE turns off tracing.

7. *PRINT

requests the debugger to copy everything displayed currently on the 2260 screen face into the SYSPRINT data set (this same is overridable—see the *DDNAME command). In this way, the user may keep a history of his debugging sessions and also develop a hardcopy trail of errors for later analysis. This command does not exist in the 1052 version.

8. (1) *BREAK=name
(2) *BREAK=name/DEL
(3) *BREAK=/DEL
(4) *BREAK=name/loc
(5) *BREAK=name/loc/verify string

instructs the debugger to insert a breakpoint (cases 1, 4, and 5) or delete a breakpoint (cases 2 and 3). In the former case, a breakpoint, with the given name, is inserted at a specified location. In case 1, it is inserted at the last displayed position: in case 4, at the named location; and in case 5, at the named location-after DYDE has verified that the supplied string (in hex) matches the information that is actually in core at that location. If the two strings do not match, the location is displayed, but no breakpoint is inserted nor is any other change made. Case 2 tells the debugger to delete the named breakpoint; and case 3 tells the debugger to delete the current breakpoint (if one exists).

9. *GO

instructs the debugger to execute (or resume) the current program. If this is the first *GO issued after an *NAME, the program begins at the link-editor-assigned entry point. If the program is halted currently at a breakpoint, control is resumed at the breakpoint's location unless an *RESUME has modified this address. If the program has program checked (a specific type of 360 interrupt such as an invalid address specification), the only way to resume it without reloading a fresh copy is thro ıgh the *RESUME.

10. *RESUME=loc

specifies that when program execution is restarted, the debugger should resume execution at the specified address rather than starting at the current breakpoint. This is the only way to resume a program that has program checked. Note that great care must be exercised when using this command to guarantee that registers and program cells are properly set so that another program check does not cocur.

11. *DUMP

tells the debugger to dump itself and the program as if an ABEND (an abnormal termination SVC with the code of 100) were located at the current breakpoint rather than the machine instruction actually there.

12.
(1) *MODIFY='COND'/value
(2) *MODIFY=loc/value
(3) *MODIFY=reg no/value
(4) *MODIFY=value
(5) *MODIFY=loc/rep value/verify value

instructs the debugger to modify the program being debugged. In cases 1 and 3, the debugger modifies either the condition code set when the program resumes or the value of the specified register. For the condition code, the user supplies the mask as if he were testing it—*MODIFY='COND'/8 would cause the instruction BC 8 to branch, whereas BC 7 would not. For the register, the hex digits supplied replace the same number of digits in the register—if register 3 contains ABCD1234 and if the command *MODIFY= #3/0000 were issued, the new value would be 00001234. In case 2, the specified location is modified by the supplied value; in case 5, the specified location is modified by the rep value, after comparing it with the verify value; and in case 4, the last displayed location is modified. All hex digits supplied are modified in all cases; i.e., if location 1000 contained 47F0,1234 and if the command *MODIFY=47AF were issued, the new value would be 47AF,1234. Note that in cases 2 and 4 the value supplied may contain imbedded commas.

13.
(1) *CSECT=loc
(2) *CSECT

defines a new context for the evaluation of expressions used for the loc parameters. In case 1, the location specified is used as the new base. Case 2 resets the program's base to the first byte of the load module.

Several previous commands contain a location specification as a parameter (signified by loc in the command's syntax). Wherever this is required, the user may code the sum or difference of any combination of the following elements:

1. ?hex value—hex displacement from the current base point (see *CSECT);
2. &hex value—absolute displacement from the first addressable byte in the machine;
3. decimal value—decimal displacement from the current base point (see *CSECT);
4. *–location of the current breakpoint;
5. # followed by a register (i.e., #3);
6. character string—absolute location of the specified symbol;
7. any sum or difference of the above enclosed in parentheses (no limit on the depth)—meaning the contents of the expression within the parentheses.

Cases 1, 2, and 7 require further explanation. When the program is loaded initially, all displacements are evaluated with reference to the first byte of the load module. This is independent of the linkage-editor-assigned entry point. Thus, ?44 refers to 68 (decimal) bytes after the first byte of the load module. The *CSECT command may be used to modify this; i.e., if an *CSECT=?44 is issued, the reference to ?44 refers to a location 136 (decimal) from the entry point. In this way, the user may move from one control section to another without having to compute displacement plus linkage-editor-assigned control section address. This feature may be used when, for example, one program dynamically loads another. The user may plant a breakpoint just before the actual transfer of control, discover the location of the entry point of the LOADed program (it should be in a register), and plant a breakpoint there (perhaps using the *BREAK=/(#15) command). When the second breakpoint is reached, the user may issue a *CSECT =* command to set the context to the LOADed program.*

Examples of valid loc parameters follow:

1. (((&10))+4) would locate the current TCB (location x'10' in the machine contains the address of the communications vector table; the first word points to a double word in core, and the second word contains the address of the current TCB).
2. If register 3 contained the value x'10', ((((#3)))+4) would accomplish the same thing. If cell CVTLOC in the user's program contained the value x'10', ((((CVTLOC)))+4) would also locate the current TCB.
3. SAVE+4 should specify a location 4 bytes after the symbol # SAVE.
4. (#15) would specify the location pointed to by register 15.

The other general category of commands requests displays of items or status about the program being debugged. These do not begin with an asterisk followed by a keyword, but are merely commands that specify what is to be displayed. These commands follow:

* In this case, the symbol table is unavailable for the LOADed program.

1. (1) 'R'
   (2) # followed by register number

requests the debugger to display either the contents of all registers (case 1) or only the specified register (case 2). For the 2260 version of DYDE, either one line is written for a single register display or four lines, each containing the contents of four registers, are written. For the 1052 version, case 1 calls for writing three messages to the operator (without reply) for registers 0 through 11 and one WTOR (which forms the basis for the next command) for the remaining four registers. In either case and for either version, the registers are displayed as they were at the last breakpoint, including any subsequent manual modification (or all zero if the program has not yet begun execution).

2. 'COND'

requests a display of the curren-tcondition code as a decimal value between 0 and 8; i.e., if the condition code is displayed as 8, a BC 8 will branch but a BC 7 will not.

3. 'BREAK'

requests a display of the current breakpoint information. All data regarding currently active breakpoints are displayed as well as identification of the current breakpoint.

4. (1) loc
   (2) loc/length
   (3) loc//modifier
   (4) loc/length/modifier

causes the display of a particular location (see the loc parameter discussion above), and defines a 'current' location to be used if the next *MODIFY or *BREAK does not specify an explicit one. If no length or modifier information is supplied and the loc specification contains no symbol, a 4-byte hexadecimal value is displayed. If a symbol is present, its length and type attributes are used. A length, which must be a decimal less than 32, determines how many digits will form the final display. The modifier may be C, B, or R or it may be omitted. If C is coded, the value will be displayed as characters; B requests the display as a bit string of ones and zeros; and R requests a display relative to the current base point. However, if R is qualified by some value in parentheses (e.g., loc//R (BASE2)), the displayed value is relative to the value of BASE2.

One other command to DYDE exists: the asynchronous interrupt to the user's executing program. After a user has indicated his desire to resume execution of his program, DYDE does not receive control again until another breakpoint is encountered. However, if the user provides an asynchronous interrupt (by simultaneously depressing the enter and shift keys on the 2260), DYDE is given control by OS, interrupting the program being debugged (which is currently executing). DYDE plants a breakpoint where the program will resume and then terminates interrupt processing. When OS resumes the program, this breakpoint is executed, and DYDE is entered. In this manner, the user, after requesting resumption of his program, may interrupt it from the console and use all of DYDE's facilities.

## Symbol table

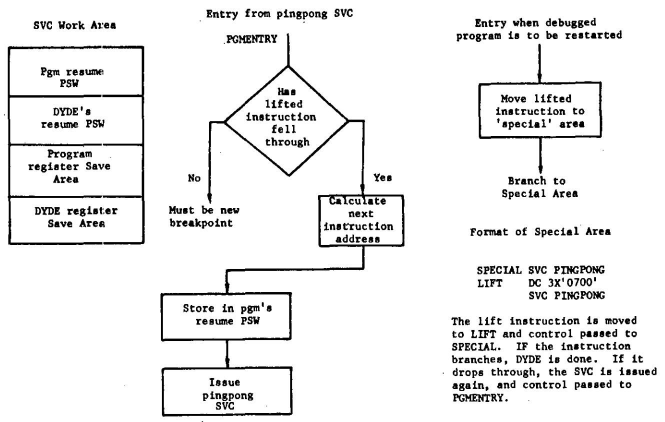
To allow the user to make symbolic references to his program, DYDE uses the OS TESTRAN facility to provide a symbol table. The assembler's test option tells it to provide the symbol table as part of its output object module. Similarly, the linkage-editor's test option tells it to write a composite symbol table (a concatenation of each symbol table present in the input load or object modules) along with the load module. Under normal processing this symbol table is ignored; i.e., when a load module is brought into core, the symbol table is stripped off. However, before loading a program in response to an *NAME command, DYDE checks the disk data set containing the program for a symbol table. If the load module on the disk does not contain symbol table entries, it is simply loaded into core, and the user is informed that symbols are not available.

However, if symbol table entries are present, they are read into core; an index is built through a hash technique; and they are written into the SYSUT1 data set. Each symbol used is present along with its attributes of type and length and its displacement. The composite external symbol dictionary (CESD) of control sections, produced by the linkage-editor, is used to build a map of the program so that each symbol may be assigned an address relative to the load point rather than a displacement from its control-section origin. As each symbol is retrieved, the first four characters are multiplied by the last four, and the middle seven bits of the resulting 64-bit product are used to index a 128-entry hash table. Each table entry contains an index to a block of data on external storage and a displacement within that block. All symbols with the same hash entry are chained together, each pointing to the block and displacement of the next symbol. Each block contains enough space for 200 symbols; the most recently referenced block is kept in core to minimize disk accesses. This method seems to work efficiently for the on-line user expecting rapid response.

## User SVC

One major deficiency of the 3600 hardware, which any debugging system must overcome, is the requirement that any transfer of control be accompanied by the setting (and the destruction) of one of the sixteen general-purpose registers. Thus, the transfer of control from the debugged program at breakpoints cannot be accomplished merely by a branch, but must be performed by an instruction that is independent of register settings. The most likely candidate is a supervisor call (SVC) and its associated supervisor call routine, which can arrange for saving all sixteen registers and the transfer of control. However, the modification of the user's program when such an SVC is inserted to represent a breakpoint requires that destroyed instructions be executed interpretively out of line, if the breakpoint is to be used in the future. This is quite expensive since approximately 120 instructions are in the 360 repertoire, and each one's interpretation must be coded separately. Using the EXECUTE instruction to execute the one modified instruction out of line is another possibility. However, this requires that all sixteen registers be properly set before the EXECUTE instruction is issued, and that control be transferred to the next instruction in the program without destroying any register contents.

To solve this problem, DYDE employs a type III user-written supervisor call that allows both DYDE and the program to be debugged to reside as "co-routines" in the same job. This SVC can be viewed from the outside as having a pingpong effect on the control flow. Each time the SVC is issued, after an initial call, control is passed to the other co-routine; i.e., the first call passes to the SVC routine an address within DYDE for register and program-status-word (PSW) save areas, one for itself and one for the program being debugged. Thereafter, each issuance saves the registers and PSW of the issuing co-routine in its area and restores the registers and PSW of the other member of the pair. Thus, each breakpoint inserted in the program being debugged calls for DYDE to


SVC Work Area
Pgm resume PSW
DYDE's resume PSW
Program register Save Area
DYDE register Save Area
Entry from pingpong SVC
PGMENTRY
Has lifted instruction fell through
No
Yes
Must be new breakpoint
Calculate next instruction address
Store in pgm's resume PSW
Issue pingpong SVC
Entry when debugged program is to be restarted
Move lifted instruction to 'special' area
Branch to Special Area
Format of Special Area

```
SPECIAL SVC PINGPONG
LIFT    DC 3X'0700'
        SVC PINGPONG
```

The lift instruction is moved to LIFT and control passed to SPECIAL. IF the instruction branches, DYDE is done. If it drops through, the SVC is issued again, and control passed to PGMENTRY.

Figure 3—User program—DYDE interfaces

lift and save the current instruction at that location and to plant the two-byte SVC. When the SVC is executed, control passes to DYDE at an entry point specified by it; a note is made of the location where the SVC was issued. When the user indicates he wants his program resumed, the lifted instruction is moved into a special area in DYDE; the program's resume address is updated to point to this location; and DYDE issues the SVC. This causes the program's registers to be restored and control to be passed to the lifted instruction. If it is a branch, control passes directly back to the program. However, if it is not a branch, control will pass to the next instruction in this special area, which happens to be another pingpong SVC call. Since it was issued while the program was in execution, control is passed to DYDE, which notes that the SVC was issued from within its own address space and that the lifted instruction dropped through. DYDE then calculates the address of the instruction following the lifted instruction, places it in the program's resume-program-status word, and reissues the SVC. This causes control to return to the program, which remains in control until another breakpoint is reached (see Figure 3). The only instructions that cannot be executed when moved are the Branch and Link and the Branch and Link Register, which are location dependent—they load a specific register with the current contents of the location counter and then branch to another location. DYDE interprets both instructions.

## APPENDIX A

### *1052 Operation*

The 1052 is the normal OS operator's console. DYDE uses the Write to Operator (WTO) and the Write to Operator with Reply (WTOR) facilities to communicate with the user. These macros allow any program to type a message on the typewriter, or to type a message and wait for a reply. This facility provides a very rudimentary form of interaction; not only is the typewriter slow, but the form of user commands is, of necessity, burdensome. More importantly, the console is used by OS for communications with the operator. As such, it types out not only declarative but informative messages and expects some replies. Thus, a user wishing to use DYDE on a 1052 must tolerate other console activity; separate those messages sent to him by DYDE from other operator messages, usually by noting the message content; and tag his commands with the number of the message to which he is replying.

The mechanism for these replies is bothersome. The user first depresses the REQUEST key, then, when the system responds with the proceed light, he must type the character R (short for REPLY), leave a space, and then type the following: (1) the number of the outstanding message to which he is replying, (2) a quote, (3) the message body, (4) a terminal quote, and (5) the end of block. Assuming the user has received a proceed light, and is replying to message 3, he must type:

R 03, 'THIS IS AN EXAMPLE.'

followed by an end of block.

Using this operation, DYDE initially types out a READY message and waits for a reply. The user responds to this message using the reply mechanism—by issuing a legal command, and being careful to note the number (or tag) associated with the READY message. DYDE responds to each request with a message. If the request requires more than one line, at least one WTO is issued, with no wait for reply; it is followed by a WTOR and a wait for reply. In this manner, DYDE can debug a program that resides as one of many jobs in a multiprogrammed environment, and still keep the interference with normal system operations at a minimum.

## APPENDIX B

### *Command abbreviations*

The following command abbreviations are available:

| *Abbreviation* | *Full Form* |
|---|---|
| *NA | *NAME |
| *M | *MODIFY |
| *BR | *BREAK |
| *FI | *FINISH |
| *DD | *DDNAMES |
| *CS | *CSECT |
| *RE | *RESUME |
| *TR | *TRACE |
| *S | *SETMODE |
| null command (i.e., just the enter symbol) | if mode is next, then *NEXT if mode is not next, then *GO |

# The multics PL/1 compiler

*by* R. A. FREIBURGHOUSE

*General Electric Company*
Cambridge, Massachusetts

## INTRODUCTION

The Multics PL/1 compiler is in many respects a "second generation" PL/1 compiler. It was built at a time when the language was considerably more stable and well defined than it had been when the first compilers were built.[1,2] It has benefited from the experience of the first compilers and avoids some of the difficulties which they encountered. The Multics compiler is the only PL/1 compiler written in PL/1 and is believed to be the first PL/1 compiler to produce high speed object code.

### The language

The Multics PL/1 language is the language defined by the IBM "PL/1 Language Specifications" dated March, 1968.[1] At the time this paper was written most language features were implemented by the compiler but the run time library did not include support for input and output, as well as several lesser features. Since the multi-tasking primitives provided by the Multics operating system were not well suited to PL/1 tasking, PL/1 tasking was not implemented. Inter-process communication (Multics tasking) may be performed through calls to operating system facilities.

### The system environment

The compiler and its object programs operate within the Multics operating system.[3,4,5] The environment provided by this system includes a virtual two dimensional address space consisting of a large number of segments. Each segment is a linear address space whose addresses range from 0 to 64K. The entire virtual store is supported by a paging mechanism which is invisible to the program. Each program operating in this environment consists of two segments: a text segment containing a pure re-entrant procedure, and a linkage segment containing out-references (links), definitions (entry names), and static storage local to the program. The text segment of each program is sharable by all other users on the system. Linking to a called program is normally done dynamically during program execution.

### Implementation techniques

The entire compiler and the Multics operating system were written in EPL, a large subset of PL/1 containing most of the complex features of the language. The EPL compiler was built by a team headed by M. D. McIlroy and R. Morris of Bell Telephone Laboratories. Several members of the Multics PL/1 project modified the original EPL compiler to improve its object code performance, and utilized the knowledge acquired from this experience in the design of the Multics PL/1 compiler. EPL and Multics PL/1 are sufficiently compatible to allow the Multics PL/1 compiler to compile itself and the operating system.

The Multics PL/1 compiler was built and de-bugged by four experienced system programmers in 18 months. All program preparation was done on-line using the CTSS time-sharing system at MIT. Most de-bugging was done in a batch mode on the GE645, but final de-bugging was done on-line using Multics.

The extremely short development time of 18 months was made possible by these powerful tools. The same design programmed in a macro-assembly language using card input and batched runs would have required twice as much time, and the result would have been extremely unmanageable.

## Design objectives

The project's design decisions and choice of techniques were influenced by the following objectives:

1. A correct implementation of a reasonably complete PL/1 language.
2. A compiler which produced relatively fast object code for all language constructs. For similar language constructs, the object code was expected to equal or exceed that produced by most Fortran or COBOL compilers.
3. Object program compatibility with EPL object programs and other Multics languages.
4. An extensive compile time diagnostic facility.
5. A machine independent compiler capable of bootstrapping itself onto other hardware.

The compiler's size and speed were considered less important than the above mentioned objectives. Each phase of the original compiler occupies approximately 32K, but after the compiler has compiled itself that figure will be about 24K. The original compiler was about twice as slow as the Multics Fortran compiler. The bootstrapped version of the PL/1 compiler is expected to be considerably faster than the original version but it will probably not equal the speed of Fortran.

### *An overview of the compiler*

The Multics PL/1 compiler is designed along traditional lines. It is not an interactive compiler nor does it perform partial compilations. The compiler translates PL/1 external procedures into relocatable binary machine code which may be executed directly or which may be bound together with other procedures compiled by any Multics language processor.

The notion of a phase is particularly useful when discussing the organization of the Multics PL/1 compiler. A phase is a set of procedures which performs a major logical function of compilation, such as syntactic analysis. A phase is not necessarily a memory load or a pass over some data base although it may, in some cases, be either or both of these things.

The dynamic linking and paging facilities of the Multics environment have the effect of making available in virtual storage only those specific pages of those particular procedures which are referenced during an execution of the compiler. A phase of the Multics PL/1 compiler is therefore only a logical grouping of procedures which may call each other. The PL/1 compiler is organized into five phases: Syntactic Translation, Declaration Processing, Semantic Translation, Optimization, and Code Generation.

## The internal representation

The internal representation of the program being compiled serves as the interface between phases of the compiler. The internal representation is organized into a modified tree structure (the program tree) consisting of nodes which represent the component parts of the program, such as blocks, groups, statements, operators, operands, and declarations. Each node may be logically connected to any number of other nodes by the use of pointers.

Each source program block is represented in the program tree by a block node which has two lists connected to it: a statement list and a declaration list. The elements of the declaration list are symbol table nodes representing declarations of identifiers within that block. The elements of the statement list are nodes representing the source statements of that block. Each statement node contains the root of a computation tree which represents the operations to be performed by that statement. This computation tree consists of operator nodes and operand nodes.

The operators of the internal representation are n-operand operators whose meaning closely parallels that of the PL/1 source operators. The form of an operand is changed by certain phases, but operands generally refer to a declaration of some variable or constant. Each operand also serves as the root of a computation tree which describes the computations necessary to locate the item at run time.

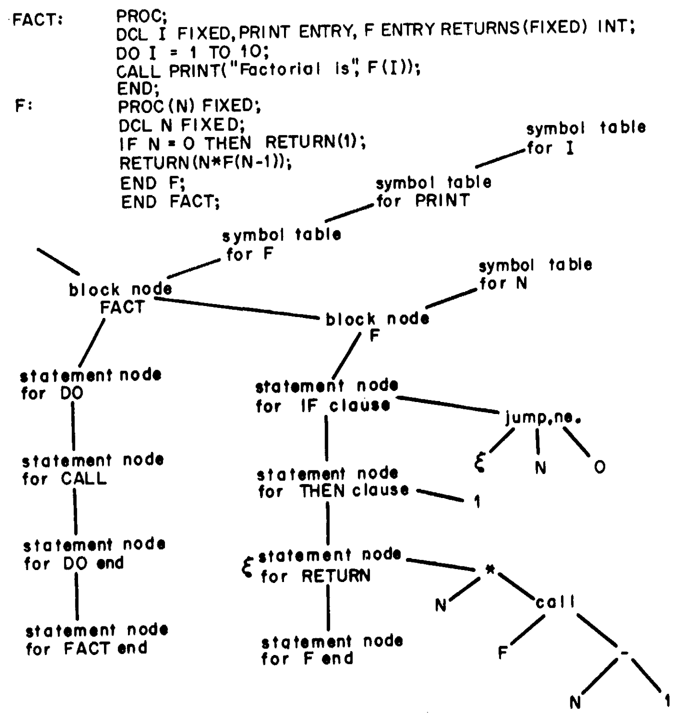
This internal representation is machine independent in that it does not reflect the instruction set, the addressing properties, or the register arrangement of the GE645. The first four phases of the compiler are also machine independent since they deal only with this machine independent internal representation. Figure 1 shows the internal representation of a simple program.

### *Syntactic translation*

Syntactic analysis of PL/1 programs is slightly more difficult than syntactic analysis of other languages such as Fortran. PL/1 is a larger language containing more syntactic constructs, but it does not present any significantly new problems. The syntactic translator consists of two modules called the lexical analyzer and the parse.

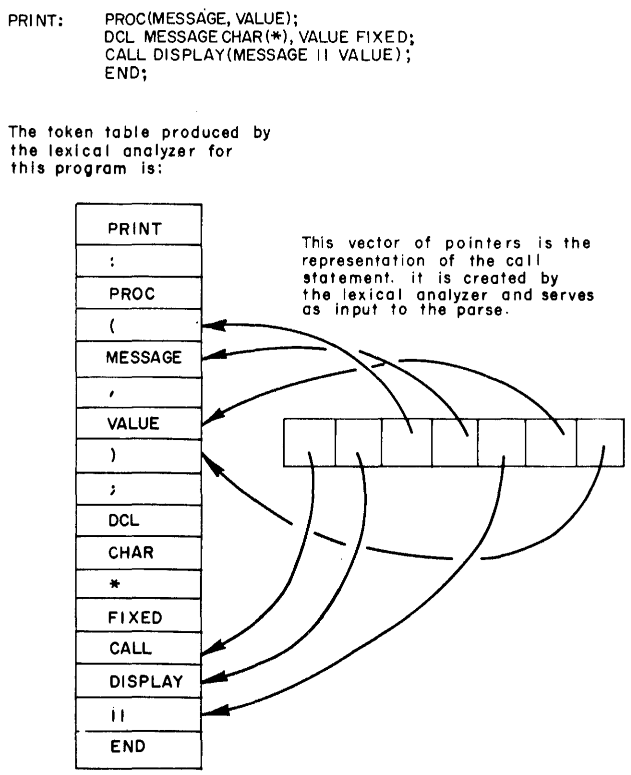
## Lexical analysis

The lexical analyzer organizes the input text into groups of tokens which represent a statement. It also creates the source listing file and builds a token table which contains the source representation of all tokens in



```
FACT:   PROC;
        DCL I FIXED,PRINT ENTRY, F ENTRY RETURNS(FIXED) INT;
        DO I = 1 TO 10;
        CALL PRINT("Factorial is", F(I));
        END;
F:      PROC(N) FIXED;
        DCL N FIXED;
        IF N = 0 THEN RETURN(1);
        RETURN(N*F(N-1));
        END F;
        END FACT;
```

Figure 1—The internal representation of a program. The example is greatly simplified. Only the statements of procedure *F* are shown in detail.



```
PRINT:  PROC(MESSAGE, VALUE);
        DCL MESSAGE CHAR(*), VALUE FIXED;
        CALL DISPLAY(MESSAGE II VALUE);
        END;
```

The token table produced by the lexical analyzer for this program is:

PRINT
:
PROC
(
MESSAGE
,
VALUE
)
;
DCL
CHAR
*
FIXED
CALL
DISPLAY
II
END
This vector of pointers is the representation of the call statement. it is created by the lexical analyzer and serves as input to the parse.

Figure 2—The output of the lexical analyzer.

the source program. A token is an identifier, a constant, an operator or a delimiter. The lexical analyzer is called by the parse each time the parse wants a new statement.

The lexical analyzer is an approximation to a finite state machine. Since the lexical analyzer must produce output as well as recognize tokens, action codes are attached to the state transitions of the finite state machine. These action codes result in the concatenation of individual characters from the output until a recognized token is formed. Constants are not converted to their internal format by the lexical analyzer. They are converted by the semantic translator to a format which depends on the context in which the constant appears.

The token table produced by the lexical analyzer contains a single entry for each unique token in the source program. Searching of the token table is done utilizing a hash coded scheme which provides quick access to the table. Each token table entry contains a pointer which may eventually point to a declaration of the token. For each statement, the lexical analyzer builds a vector of pointers to the tokens which were found in the statement. This vector serves as the input to the parse. Figure 2 shows a simple example of lexical analysis.

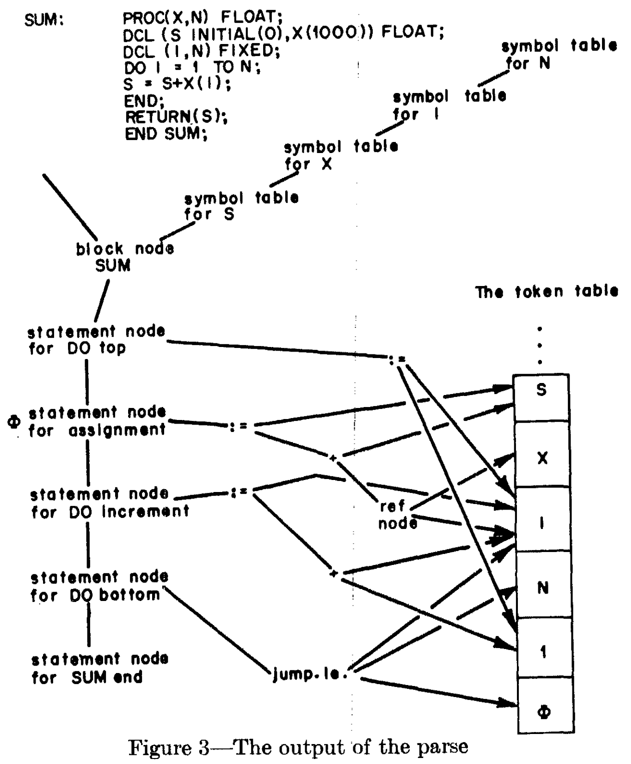
## The parse

The parse consists of a set of possibly recursive procedures, each of which corresponds to a syntactic unit of the language. These procedures are organized to perform a top down analysis of the source program. As each component of the program is recognized, it is transformed into an appropriate internal representation. The completed internal representation is a program tree which reflects the relationships between all of the components of the original source program. Figure 3 shows the results of the parse of a simple program.

Syntactic contexts which yield declarative information are recognized by the parse, and this information is passed to a module called the context recorder which constructs a data base containing this information. Declare statements are parsed into partial symbol table nodes which represent declarations.

## The problem of backup

The top down method of syntactic analysis is used because of its simplicity and flexibility. The use of a simple statement recognition algorithm made it possible



```
SUM:    PROC(X,N) FLOAT;
        DCL (S INITIAL(0),X(1000)) FLOAT;
        DCL (I,N) FIXED;
        DO I = 1 TO N;
        S = S+X(I);
        END;
        RETURN(S);
        END SUM;
```

symbol table for N
symbol table for I
symbol table for X
symbol table for S
block node SUM
The token table
statement node for DO top
Φ statement node for assignment
statement node for DO increment
statement node for DO bottom
statement node for SUM end
:=
+
ref node
jump.le.
S
X
I
N
1
Φ

Figure 3—The output of the parse

to eliminate all backup. The statement recognizer identifies the type of each statement before the parse of that statement is attempted. The algorithm used by this procedure first attempts to recognize assignment statements using a left to right scan which looks for token patterns which are roughly analogous to $X =$ or $X\ (\ \ ) =$. If a statement is not recognized as an assignment, its leading token is matched against a keyword list to determine the statement type. This algorithm is very efficient and is able to positively identify all legal statements without requiring keywords to be reserved.

*Declaration processing*

PL/1 declaration processing is complicated by the great variety of data attributes and by the context sensitive manner in which they are derived. Two modules, the context processor and the declaration processor, process declarative information gathered by the parse.

### The context processor

The context processor scans the data base containing contextually derived attributes produced during the parse by the context recorder. It either augments the partial symbol table created from declare statements or creates new declarations having the same format as those derived from declare statements. This activity creates contextual and implicit declarations.

### The declaration processor

The declaration processor develops sufficient information about the variables of the program so that they may be allocated storage, initialized and accessed by the program's operators. It is organized to perform three major functions: the preparation of accessing code, the computation of each variable's storage requirements, and the creation of initialization code.

The declaration processor is relatively machine independent. All machine dependent characteristics, such as the number of bits per word and the alignment requirements of data types, are contained in a table. All computations or statements produced by the declaration processor have the same internal representation as source language expressions or statements. Later phases of the compiler do not distinguish between them.

#### The use of based references by the declaration processor

The concept of a based reference is useful to the understanding of PL/1 data accessing and the implementation of a number of language features. A based declaration of the form *DCL A BASED* is referenced by a based reference of the form $P \rightarrow A$, where $P$ is a pointer to the storage occupied by a value whose description is given by the declaration of $A$. Multiple instances of data having the characteristics of $A$ can be referenced through the use of unique pointers, i.e., $Q \rightarrow A$, $R \rightarrow A$, etc.

The declaration processor implements a number of language features by transforming them into suitable based declarations. Automatic data whose size is variable is transformed into a based declaration.

For example the declaration:

```
DCL A(N) AUTO;
```

becomes

```
DCL A(N) BASED(P);
```

where: $P$ is a compiler produced pointer which is set upon entry to the declaring block.

Based declarations are also used to implement parameters. For example,

```
X: PROC (C); DCL C;
```

becomes

```
X: PROC (P); DCL C BASED(P);
```

where: $P$ is a pointer which points to the argument corresponding to the parameter $C$.

### Data accessing

The address of an item of PL/1 data consists of three basic parts: a pointer to some storage location, a word offset from that location and a bit offset from the word offset. Either or both offsets may be zero. The term "word" is understood to refer to the addressable unit of a computer's storage.

Example 1

```
DCL A AUTO;
```

The address of $A$ consists of a pointer to the declaring block's automatic storage, a word offset within that automatic storage and a zero bit offset

Example 2

```
DCL 1 S BASED(P),
      2 A BIT(5),
      2 B BIT(N)
```

When referenced by $P \rightarrow B$, the address of $B$ is a pointer $P$, a zero word offset and a bit offset of 5. The word offset may include the distance from the origin of the item's storage class, as was the case with the first example, or it may be only the distance from the level-one containing structure, as it was in the last example. The term "level-one" refers to all variables which are not contained within structures. Subscripted array element references, $A(K, J)$, or sub-string references, $SUBSTR(X, K, J)$, may also be expressed as offsets.

### Offset expressions

The declaration processor constructs offset expressions which represent the distance between an element of a structure and the data origin of its level-one containing structure. If an offset expression contains only constant terms, it is evaluated by the declaration processor and results in a constant addressing offset. If the offset expression contains variable terms, the expression results in the generation of accessing instructions in the object program. The discussion which follows describes the efficient creation of these offset expressions.

Given a declaration of the form:

```
DCL 1 S,
      2 A BIT(M),
      2 B BIT(5),
      2 C FLOAT;
```

The offset of $A$ is zero, the offset of $B$ is $M$ bits, and the offset of $C$ is $M + 5$ bits rounded upward to the nearest word boundary.

In general, the offset of the nth item in a structure is:

$$b_n(c_{n-1}(s_{n-1}) + b_{n-1}(c_{n-2}(s_{n-2}) + b_{n-2} (\cdots b_3(c_2(s_2)) + b_2(c_1(s_1)))\cdots)))$$

where: $b_k$ is a rounding function which expresses the boundary requirement of the *kth* item.

$s_k$ is the size of the *kth* item.

$c_k$ is the conversion factor necessary to convert $s_k$ to some common units such as bits.

The declaration processor suppresses the creation of unnecessary conversion functions ($c_k$) and boundary functions ($b_k$) by keeping track of the current units and boundary as it builds the expression. As a result the offset expressions of the previous example do not contain conversion functions and boundary functions for $A$ and $B$.

During the construction of the offset expression, the declaration processor separates the constant and variable terms so that the addition of constant terms is done by the compiler rather than by accessing code in the object program. The following example demonstrates the improvement gained by this technique.

```
DCL 1 S,
      2 A BIT(5),
      2 B BIT(K),
      2 C BIT(6),
      2 D BIT(10);
```

The offset of $D$ is $K+11$ instead of $5+K+6$.

The word offset and the bit offset are developed separately. Within each offset, the constant and variable parts are separated. These separations result in the minimization of additions and unit conversions. If the declaration contains only constant sizes, the resulting offsets are constant. If the declaration contains expressions, then the offsets are expressions containing the minimum number of terms and conversion factors.

The development of size and offset expressions at compile time enables the object program to access data without the use of data descriptors or "dope vectors."[6] Most existing PL/1 implementations make extensive use of such descriptors to access data whose size or offsets are variable. Unless these descriptors are implemented by hardware, their use results in rather inefficient object code. The Multics PL/1 strategy of developing offset expressions from the declarations results in accessing code similar to that produced for subscripted array references. This code is generally more efficient than code which uses descriptors.

In general, the offset expressions constructed by the declaration processor remain unchanged until code generation. Two cases are exceptions to this rule: subscripted array references, $A(K,J)$, and sub-string references, $SUBSTR(X,K,J)$. Each subscripted reference or sub-string reference is a reference to a unique sub-datum within the declared datum and, therefore, requires a unique offset. The semantic translator constructs these unique offsets using the subscripts from the reference and the offset prepared by the declaration processor.

### Allocation

The declaration processor does not allocate storage for most classes of data, but it does determine the amount of storage needed by each variable. Variables are allocated within some segment of storage by the code generator. Storage allocation is delayed because, during semantic translation and optimization, additional declarations of constants and compiler created variables are made.

### Initialization

The declaration processor creates statements in the prologue of the declaring block which will initialize automatic data. It generates DO statements, IF statements and assignment statements to accomplish the required initialization.

The expansion of the initial attribute for based and controlled data is identical to that for automatic data except that the required statements are inserted into the program at the point of allocation rather than in the prologue.

Since array bounds and string sizes of static data are required by the language to be constant, and since all values of the initial attribute of static data must be constant, the compiler is able to initialize the static data at compile time. The initialization is done by the code generator at the time it allocates the static data.

## *Semantic translation*

The semantic translator transforms the internal representation so that it reflects the attributes (semantics) of the declared variables without reflecting the properties of the object machine. It makes a single scan over the internal representation of the program. A compiler, which had no equivalent of the optimizer phase and which did not separate the machine dependencies into a separate phase, could conceivably produce object code during this scan.

### Organization of the semantic translator

The semantic translator consists of a set of recursive procedures which walk through the program tree. The actions taken by these procedures are described by the general terms: operator transformation and operand processing. Operator transformation includes the creation of an explicit representation of each operator's result and the generation of conversion operators for those operands which require conversion. Operand processing determines the attributes, size and offsets of each operator's operands.

### Operator transformation

The meaning of an operator is determined by the attributes of its operands. This meaning specifies which conversions must be performed on the operands, and it decides the attributes of the operator's result.

An operator's result is represented in the program tree by a temporary node. Temporary nodes are a further qualification of the original operator. For example, an add operator whose result is fixed-point is a distinct operation from an add operator whose result is floating-point. There is no storage associated with temporaries—they are allocated either core or register storage by the code generator. A temporary's size is a function of the operator's meaning and the sizes of the operator's operands. A temporary, representing the intermediate result of a string operation, requires an expression to represent its length if any of the string operator's operands have variable lengths.

### Operand processing

Operands consist of sub-expressions, references to variables, constants, and references to procedure names or built-in functions. Sub-expression operands are processed by recursive use of operator transformation and operand processing. Operand processing converts constants to a binary format which depends on the context in which the constant was used. References to variables or procedure names are associated with their appropriate declaration by the search function. After the search function has found the appropriate declaration, the reference may be further processed by the subscriptor or function processor.

## The Search function

During the parse, it is not possible for references to source program variables to know the declared attributes of the variable because the PL/1 language allows declarations to follow their use. Therefore, references to source program variables are parsed into a form which contains a pointer to a token table entry rather than to a declaration of the variable. Figure 3 shows the output of the parse. The search function finds the proper declaration for each reference to a source program variable. The effectiveness of the search depends heavily on the structure of the token table and the symbol table. After declaration processing, the token table entry representing an identifier contains a list of all the declarations of that identifier. See Figure 4.

The search function first tries to find a declaration belonging to the block in which the reference occurred. If it fails to find one, it looks for a declaration in the next containing block. This process is repeated until a declaration is found. Since the number of declarations on the list is usually one, the search is quite fast. In its attempt to find the appropriate declaration, the search function obeys the language rules regarding structure qualification. It also collects any subscripts used in the reference and places them into a subscript list. Depending on the attributes of the referenced item, the subscript list serves as input to the function processor or subscriptor.

The declaration processor creates offset expressions and size expressions for all variables. These expressions, known as accessing expressions, are rooted in a reference node which is attached to a symbol table node. The reference node contains all information necessary to access the data at run time. The search function translates a source reference into a pointer to this reference node. See Figure 5.

## Subscripting

Since each subscripted reference is unique, its offset expression is unique. To reflect this in the internal representation, the subscriptor creates a unique reference node for each subscripted reference. See Figure 6. The following discussion shows the relationship between the declared array bounds, the element size, the array offset and subscripts.

```
TOP:    PROC;
        DCL B POINTER;
            BEGIN;
                DCL B FLOAT;
                    BEGIN;
                        DCL B FIXED;
                    END;
            END;
        END;
```

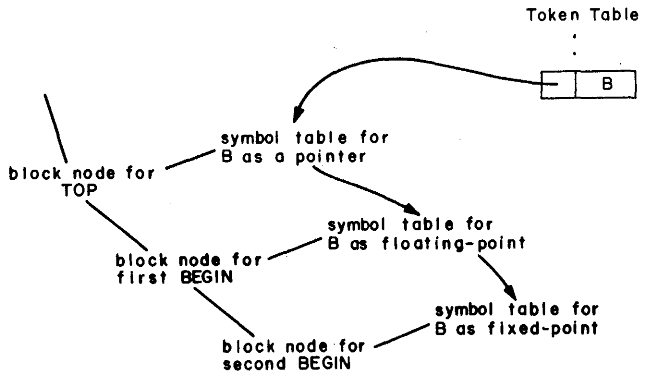

Token Table
B
block node for TOP
symbol table for B as a pointer
block node for first BEGIN
symbol table for B as floating-point
block node for second BEGIN
symbol table for B as fixed-point

Figure 4—The relationship between the token table and the symbol table

```
DEM:    PROC;
        DCL 1 S,
             2 A(N) FLOAT,
             2 B(M) FIXED;
        S.B(I) = O;
        END;
```

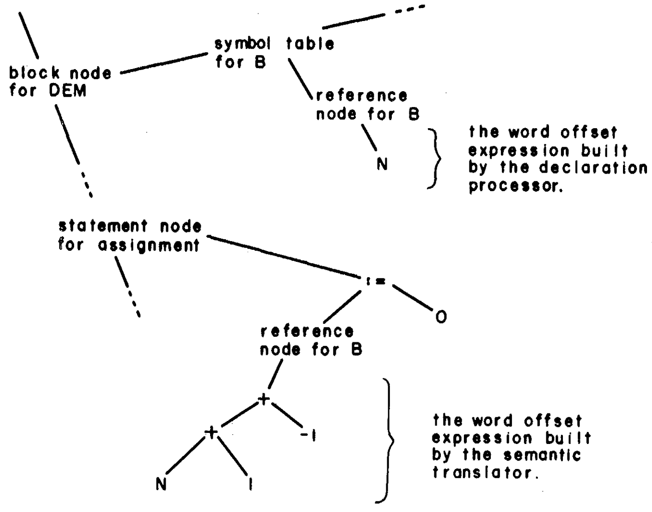

block node for DEM
symbol table for B
reference node for B
N
the word offset expression built by the declaration processor.
statement node for assignment
:=
O
reference node for B
+
+
-I
N
I
the word offset expression built by the semantic translator.

Figure 5—A simplified diagram showing the effects of subscripting

Let us consider the case of an array declared:

$$a(l_1:u_1, l_2:u_2, \cdots, l_n:u_n)$$

Its element size is $s$ and its offset is $b$.

The multipliers for the array are defined as:

$$\begin{aligned} m_n &= s \\ m_{n-1} &= (u_n - l_n + 1)s \\ m_{n-2} &= (u_{n-1} - l_{n-1} + 1)m_{n-1} \\ &\cdot \\ &\cdot \\ &\cdot \\ m_1 &= (u_2 - l_2 + 1)m_2 \end{aligned}$$

The offset of a reference $a(i_1, i_2, \cdots, i_n)$ is computed as:

$$v + \sum_{j=1}^{n} i_j m_j$$

where: $v$ is the virtual origin. The virtual origin is the offset obtained by setting the subscripts equal to zero. It serves as a convenient base from which to compute the offset of any array element.

During the construction of all expressions, the constant terms are separated from the variable terms and all constant operations are performed by the compiler. Since the virtual origin and the multipliers are common to all references, they are constructed by the declaration processor and are repeatedly used by the subscriptor.

Arrays of PL/1 structures which contain arrays may result in a set of multipliers whose units differ. The declaration:

```
DCL 1 S(10),
      2 A PTR,
      2 B(10) BIT(2);
```

yields two multipliers of different units. The first multiplier is the size of an element of $S$ in words, while the second multiplier is the size of an element of $B$ in bits.

Array parameters which may correspond to an array cross section argument must receive their multipliers from an argument descriptor. Since the arrangement of the cross section elements in storage is not known to the called program, it cannot construct its own multipliers and must use multipliers prepared by the calling program. Note that the current definition of PL/1 allows any array parameter to receive a cross section argument.

```
FIG:    PROC;
        DCL (X, Y, Z) FLOAT;
        X = Y+Z;
        END;
```

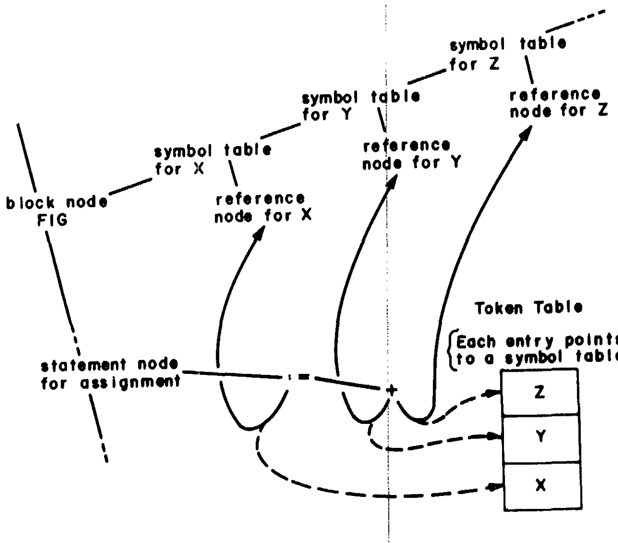


Figure 6—The internal representation of a statement before and after the execution of the search function. The broken lines show the statement's operands before the search

## The function processor

An operand which is a reference to a procedure is expanded by the function processor into a call operator and possible conversion operators. Built-in function references result in new operators or are translated into expressions consisting of operators and operands.

## Generic procedure references

A generic entry name represents a family of procedures whose members require different types of arguments.

```
DCL ALPHA GENERIC  (BETA
                    ENTRY(FIXED)),
                    GAMMA
                    ENTRY(FLOAT));
```

A reference to *ALPHA* (*X*) will result in a call to *BETA* or *CAMMA* depending on the attributes of *X*.

The declaration processor chains together all members of a generic family and the function processor selects the appropriate member of the family by matching the arguments used in the reference with the declared argument requirements of each member. When the appropriate member is found, the original reference is replaced by a reference to the selected member.

### Argument processing

The function processor matches arguments to user-declared procedures against the argument types required for the procedure. It inserts conversion operators into the program tree where appropriate, and it issues diagnostics when it detects illegal cases.

The return value of a function is processed as if it were the n+ 1th argument to the procedure, eliminating the distinction between subroutines and functions.

The function processor determines which arguments may possibly correspond to a parameter whose size or array bounds are not specified in the called procedure. In this case, the argument list is augmented to include the missing size information. A more detailed description of this issue is given later in the discussion of object code strategies.

### The built-in function processor

The built-in function processor is basically a table driven device. The driving table describes the number and kind of arguments required by each function and is used to force the necessary conversions and diagnostics for each argument. Most functions require processing which is unique to that function, but the table driven device minimizes the amount of this processing.

The *SUBSTR* built-in function is of particular importance since it is a basic PL/1 string operator. It is a three argument function which allows a reference to be made to a portion of a string variable, i.e., *SUBSTR* (*X*, *I*, *J*) is a reference to the ith through i + j − 1th character (or bit) in the string *X*.

This function is similar to an array element reference in the sense that they both determine the offsets of the reference. The processing of the *SUBSTR* function involves adjusting the offset and length expressions contained in the reference node of *X*. As is the case in all compiler operations on the offset expressions, the constant and variable terms are separated to minimize the object code necessary to access the data.

### *The optimizer*

The compiler is designed to produce relatively fast object code without the aid of an optimizing phase. Normal execution of the compiler will by-pass the optimizer, but if extensively optimized object code is desired, the user may set a compiler command option which will execute the optimizer. The optimizer consists of a set of procedures which perform two major optimizations: common sub-expression removal and removal of computations from loops. The data bases necessary for these optimizations are constructed by the parse and the semantic translator. These data bases consist of a cross-reference structure of statement labels and a tree structure representing the DO groups of each block. Both optimizations are done on a block basis using these two data bases.

Although the optimizer phase was not implemented at the time this paper was written, all data bases required by the optimizer are constructed by previous phases of the compiler and the abnormality of all variables is properly determined.

## Optimization of PL/I programs

The on-condition mechanism of the PL/1 language makes the optimization of PL/1 programs considerably more difficult than the optimization of Fortran programs. Assuming that an optimized version of a program should yield results identical to those produced by the un-optimized version, then if any on-conditions are enabled in a given region of the program, the compiler cannot remove or reorder the computations performed in that region. (Consider the case of a divide by zero on unit which counts the number of times that the condition occurs.)

Since some on-conditions are enabled by default, most PL/1 programs cannot be optimized. Because of the difficulty of determining the abnormality of a program's variables, the optimization of those programs which may be optimized requires a rather intelligent compiler. A variable is abnormal in some block if its value can be altered without an explicit indication of that fact present in that block. An optimizing PL/1 compiler must consider all based variables, all arguments to the *ADDR* function, all defined variables, and all base items of defined variables to be abnormal. If the compiler expects values of variables to be retained throughout the execution of a call, it must also consider all parameters, all external variables, and all arguments of irreducible functions to be abnormal.

Because of the difficulty of optimizing programs written in the current PL/1 language[1] compilers should probably not attempt to perform general optimizations but should concentrate on special case optimizations which are unique to each implementation. Future revisions to the language definition may help solve the optimization problem.

### *The code generator*

The code generator is the machine dependent portion of the compiler. It performs two major functions: it allocates data into Multics segments and it generates

645 machine instructions from the internal representation.

### Storage allocation

A module of the code generator called the storage allocator scans the symbol table allocating stack storage for constant size automatic data, and linkage segment storage for internal static data. For each external name the storage allocator creates a link (an out-reference) or a definition (an entry point) in the linkage segment. All internal static data is initialized as its storage is allocated.

Due to the dynamic linking and loading characteristics of the Multics environment, the allocation and initialization of external static storage is rather unusual. The compiler creates a special type of link which causes the linker module of the operating system to create and initialize the external data upon first reference. Therefore, if two programs contain references to the same item of external data, the first one to reference that data will allocate and initialize it.

### Code generation

The code generator scans the internal representation transforming it into 645 machine instructions which it outputs into the text segment. During this scan the code generator allocates storage for temporaries, and maintains a history of the contents of index registers to prevent excessive loading and storing of index values.

Code generation consists of three distinct activities: address computation, operator selection and macro expansion. Address computation is the process of transforming the offset expressions of a reference node into a machine address or an instruction sequence which leads to a machine address. Operator selection is the translation of operators into n-operand macros which reflect the properties of the 645 machine.

A one-to-one relationship often exists between the macros and 645 instructions but many operations (load long string, etc.) have no machine counterpart. All macros are expanded in actual 645 code by the macro expander which uses a code pattern table (macro skeletons) to select the specific instruction sequences for each macro.

*Object code strategies*

### The object code design

The design of the object code is a compromise between the speed obtainable by straight in-line code and the necessity to minimize the number of page faults caused by large object programs.

The length of the object program is minimized by the extensive use of out-of-line code sequences. These out-of-line code sequences represent invariant code which is common to all Multics PL/1 object programs. Although the compiled code makes heavy use of out-of-line code sequences, the compiled code is not in any respect interpretive. The object code produce for each operator is very highly tailored to the specific attributes of that operator.

All out-of-line sequences are contained in a single "operator" segment which is shared by all users. The in-line code reaches on out-of-line sequence through transfer instructions, rather than through the standard subroutine mechanism. We believe that the time overhead associated with the transfers is more than redeemed by the reduction in the number of page faults caused by shorter object programs. System performance is improved by insuring that the pages of the operator segment are always retained in storage.

### The stack

Multics PL/1 object programs utilize a stack segment for the allocation of all automatic data, temporaries, and data associated with on-conditions. Each task (Multics process) has its own stack which is extended (pushed) upon entry to block and is reverted (popped) upon return from a block. Prior to the execution of each statement it is extended to create sufficient space for any variable length string temporaries used in that statement. Constant size temporaries are allocated at compile time and do not cause the stack to be extended for each statement.

### Prologue and epilogue

The term prologue describes the computations which are performed after block entry and prior to the execution of the first source statement. These actions include the establishment of the condition prefix, the computation of the size of variable size automatic data, extension of the stack to allocate automatic data, and the initialization of automatic data. Epilogues are not needed because all actions which must be undone upon exit from the block are accomplished by popping the stack. The stack is popped for each return or non-local go to statement.

### Accessing of data

Multics PL/1 object code addresses all data, including members of variable sized structures and arrays directly through the use of in-line code. If the address of the data is constant, it is computed at compile time. If it is a mixture of constant and variable terms, the constant terms are combined at compile time. Descriptors are never used to address or allocate data.

### String operations

All string operations are done by in-line code or by "transfer" type subroutinized code. No descriptors or calls are produced for string operations. The *SUBSTR* built-in function is implemented as a part of the normal addressing code and is therefore as efficient as a subscripted array reference.

### String temporaries

A string temporary or dummy is designed in such a way that it appears to be both a varying and non-varying string. This means that the programmer does not need to be concerned with whether a string expression is varying or non-varying when he uses such an expression as an argument.

### Varying strings

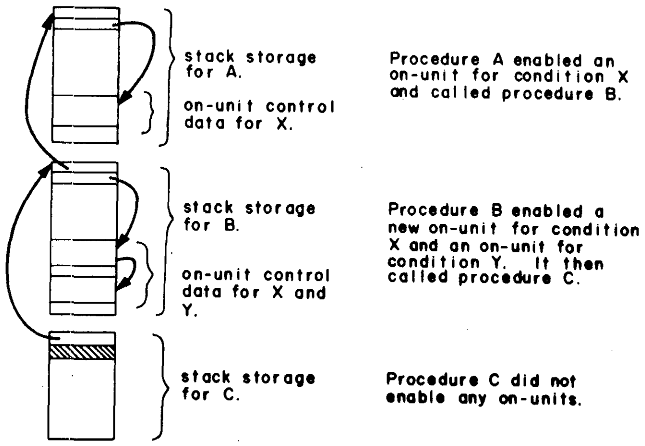
The Multics PL/1 implementation of varying strings uses a data format which consists of an integer followed by a non-varying string whose length is the declare maximum of the varying string. The integer is used to hold the current size of the string in bits or characters. Using this data format, operations on varying strings are just as efficient as operations on non-varying strings.

### On-conditions

The design of the condition machinery minimizes the overhead associated with enabling and reverting on-units and transfers most of the cost to the signal statement. All data associated with on-conditions, including the condition prefix, is allocated in the stack. The normal popping of the stack reverts all enabled on-units and restores the proper condition prefix. Stack storage associated with each block is threaded backward to the previous block. The signal statement uses this thread to search back through the stack looking for the first enabled unit for the condition being signalled. Figure 7 shows the organization of enabled on-units in the stack.

### Argument passing

The PL/1 language permits parameters to be declared with unknown array bounds or string lengths. In these cases, the missing size information is assumed to be supplied by the argument which corresponds to the parameter. This missing size information is not explicitly supplied by the programmer as is the case in Fortran, rather it must be supplied by the compiler as indicated in the following example:



stack storage for A.
on-unit control data for X.
Procedure A enabled an on-unit for condition X and called procedure B.
stack storage for B.
on-unit control data for X and Y.
Procedure B enabled a new on-unit for condition X and an on-unit for condition Y. It then called procedure C.
stack storage for C.
Procedure C did not enable any on-units.

Figure 7—Stack storage and the signal mechanism
A signal for condition X causes the signal mechanism to search back through the stack until it finds the first enabled on-unit for condition X.
An on-unit is compiled as an internal procedure. The execution of an ON-statement creates a block of on-unit control data. This control data consists of the name of the condition for which the unit was enabled and a procedure variable. The signal mechanism uses the procedure variable to invoke the on-unit. All data associated with the enabled on-unit is stored in the stack storage of the procedure which enabled it. Normal popping of the stack reverts the on-units enabled during the execution of the procedure.

```
SUB: PROC(A);          MAIN: PROC;
   .                       .
   .                       .
   .                       .
DCL A CHAR(*);         DCL SUB ENTRY;
   .
   .                   DCL B CHAR(10);
   .
                           CALL SUB(B);
                               .
                               .
                               .
```

Since parameter *A* assumes the length of the argument *B*, the compiler must include the length of *B* in the argument list of the call to *SUB*.

The declaration of an entry name may or may not include a description of the arguments required by that entry. If such a description is not supplied, then the calling program must assume that argument descriptors are needed, and must include them in all calls to the entry. If a complete argument description is contained in the calling program, the compiler can determine if descriptors are needed for calls to the entry.

In the previous example the entry *SUB* was not fully declared and the compiler was forced to assume that an argument descriptor for *B* was required. If the entry had been declared *SUB ENTRY (CHAR*(*)) the compiler could have known that the descriptor of *B* was actually required by the procedure *SUB*. Since descriptors are often created by the calling procedure but not used by the called procedure, it is desirable to separate them from the argument information which is always used by the called procedure.

Communication between procedures written in PL/1 and other languages is facilitated if the other languages do not need to concern themselves with PL/1 argument descriptors. The Multics PL/1 implementation of the argument list is shown in Figure 8. Note that the argument pointers point directly to the data (facilitating communication between languages) and that the descriptors are optional, also note that PL/1 pointers

```
TAG:    PROC;
        DCL A(10) BIT(N),B CHAR(7),C AREA(1024);
        CALL X(A,B,C);
        END;
```

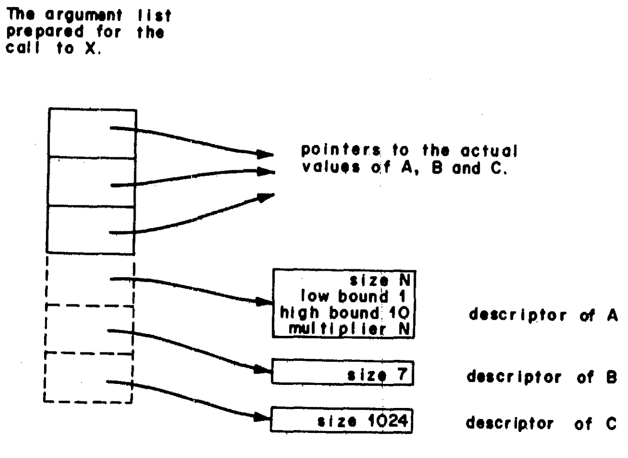


Figure 8—An argument list showing the relationship between arguments and their descriptors. The broken lines indicate that descriptors are optional.

must be capable of bit addressing in order to implement unaligned strings. Since descriptors contain no addressing information, they are quite often constant and can be prepared at compile time.

## SUMMARY

Our experiences both as users and implementors of PL/1 have led us to form a number of opinions and insights which may be of general interest.

1. It is feasible, but difficult, to produce efficient object code for the PL/1 language as it is currently defined. Unless a considerable amount of work is invested in a PL/1 compiler, the object code it generates will generally be much worse than that produced by most Fortran or COBOL compilers.
2. The difficulty of building a compiler for the current language has been seriously underestimated by most implementors. Unless the language is markedly improved and simplified this problem will continue to restrict the availability and acceptance of the language and will lead to the implementation of incompatible dialects and subsets.[7]
3. Simplification of the existing language will make it more suitable to users and implementors. We believe that the language can be simplified and still retain its "universal" character and capabilities.
4. The experience of writing the compiler in PL/1 convinced us that a subset of the language is well suited to system programming. This conviction is supported by Professor Corbato in his report on the use of PL/1 as an implementation language for the Multics system.[8] Many PL/1 concepts and constructs are valuable, but PL/1 structures and list processing seem to be the principal improvement over alternative languages.[9]

## ACKNOWLEDGMENTS

The author wishes to express recognition to members of the General Electric Multics PL/1 Project for their contributions to the design and implementation of the compiler. J. D. Mills was responsible for the design and implementation of the syntactic analyzer and the Multics system interface, B. L. Wolman designed and built the code generator and operator segment, and G. D. Chang implemented the semantic translator. Valuable advice and ideas were provided by A. H. Kvilekval. The earlier work of M. D. McIlroy and R. Morris of Bell Telephone Laboratories and numerous

persons at MIT's Project MAC provided a useful guide and foundation for our efforts.

## REFERENCES

1 *PL/1 language specifications*
Form Y33-6003-0 IBM Corp March 1968
2 The formal definition of PL/1 as specified by technical reports TR25.081, TR25.082, TR25.083, TR25.084, TR25.085, TR25.086 and TR25.087, IBM Corp Vienna Austria June 1968
3 F J CORBATO V A VYSSOTSKY
*Introduction and overview of the multics system*
Proc FJCC 1965
4 V A VYSSOTSKY F J CORBATO R M GRAHAM
*Structure of the multics supervisor*
Proc FJCC 1965
5 R C DALEY J B DENNIS
*Virtual memory, processes, and sharing in multics*
CACM Vol 11 No 5 May 1968
6 *PL/1 (F) programmer's guide*
Form C28-6594-3 IBM Corp Oct 1967
7 R F ROSIN
*PL/1 Implementation survey*
ACM SIGPLAN Notices Feb 1969
8 F J CORBATO
*PL/1 as a tool for system programming*
Datamation May 1969
9 H W LAWSON JR
*PL/1 list processing*
CACM Vol 10 No 6 June 1967

# A design for a fast computer for scientific calculations

*by* P. M. MELLIAR-SMITH

*The General Electric and English Electric Companies Limited*
Borehamwood, Hertfordshire, U. K.

Recently developed techniques, such as the associative fast store and Tomasulo's algorithm, will enable typical large scale computers to achieve 15 to 20 million instructions per second. The hardware of such machines has a very much greater potential power, but it is inefficiently used, being limited to decoding a single instruction per logic cycle. This paper proposes a technique whereby the programmer is provided with complex instructions capable of controlling the operation of the whole machine during one logic cycle. The use of such instructions for the inner loops of programs yields substantial performance improvements without significantly increased costs.

Recent efforts to develop very fast computers have generated two elegant techniques for increasing the speed of computers.

The first is the associative fast store, first used for the Titan computer at the University of Cambridge, England (a 32 word 'slave' store), and more recently for the IBM 360/85 (a 16 K byte 'buffer' store or 'cache'). The associative fast store seeks to overcome the major problem in the design of very fast computers, the disparity between the access time of suitable main stores and the potential operation time of the arithmetic units, by providing a small quantity of very fast integrated circuit store. This can be made as fast as the arithmetic units but it cannot contain more than a fraction of the information used by typical programs. However it has been found experimentally that, in any short period of time, programs do not access the whole of their storage and that a fast store, which retains a few hundred of the words most recently used by the program, is able to provide without delay almost all the information needed by the processor.

A possible method of implementation is shown in Figure 1. The fast store holds a number of words of code and data, together with their addresses. When the processor requires a particular item, the address is first sent to the fast store where it is compared simultaneously with the addresses of all the words in the fast store. Should the required item be present in the fast store, then its address will match that sent by the processor and the data can be returned to the processor with minimal delay. If none of the addresses match, then the required item must be fetched from the main store and the processor may be held up. But when the data word has been fetched, in addition to being sent to the processor, it can also be inserted into the fast store, displacing some other item, so that should it be needed again it will be immediately available.

The success of this technique is entirely dependent on the proportion of data items needed by the processor which have to be fetched from the main store, and this proportion, the failure rate, is the primary criterion of the effectiveness of the fast store. The speed of the computer is determined by:.

$$\text{Effective Access time} = \text{Fast Store Access Time} + \left(\begin{matrix}\text{Main Store}\\ \text{Access Delay}\end{matrix} * \begin{matrix}\text{Failure}\\ \text{Rate}\end{matrix}\right)$$

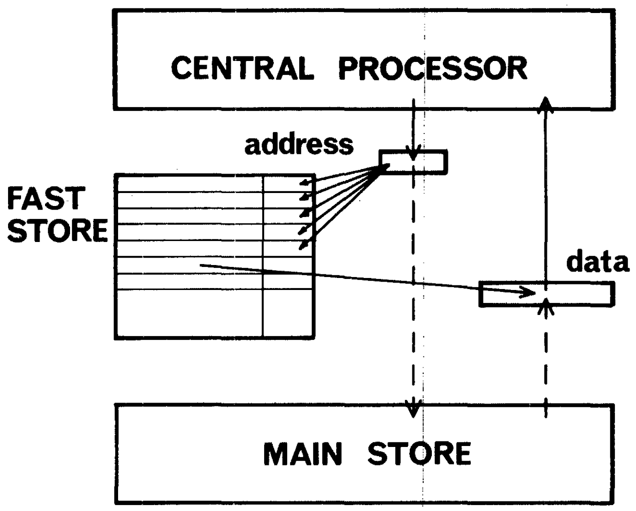


Figure 1—The use of an associative fast store to reduce the access time of the main store

If the Main Store Access Delay, which must include organisational overheads as well as the Main Store Access Time but which may be partially overlapped, is equivalent to ten fast store accesses then a failure rate of 3 percent must be attained to achieve 75 percent of the potential processor speed.

Experimental simulations with actual programs have shown that the three characteristics of the fast store which most affect the failure rates are its organisation, its size, and the size of the unit of information transferred from the main store to the fast store. The organisation of the fast store need not concern us here, except to remark that the type of organisation described above is to be preferred to alternative methods which avoid the associative access to large numbers of addresses.

The experimental simulations show that the primary method of obtaining an adequately low failure rate is to make the fast store large enough. If the fast store is smaller than several hundred words then programs refer to many items not held in the fast store and the full performance of the machine is not obtained. However the fast store must not be made too large, even without cost considerations. As the size of the fast store is increased so its access time is also inevitably increased, and eventually this increase in the physical access time of the fast store overwhelms any further reduction in the number of references to the main store. Figure 2 shows the result of simulations to obtain the effective access time of a particular integrated circuit fast store operating with a thin film main store, for seven sample programs. It can

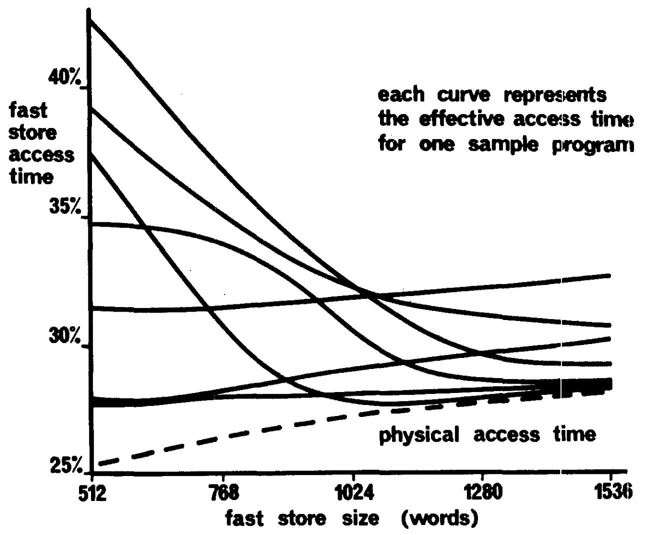


Figure 2—The physical assess time of a fast store (broken line) and the effective access times (continuous lines) for sample programs. Access time is a percentage of main store access time, storage size is in words, and line size is 4 words

be seen that for many of the programs the optimum size of the fast store is about 1000 words.

The description of a fast store given above assumed that the unit of information held in the fast store was a single word, and that information is transferred from main store in single words. The experimental simulations have shown that a more efficient unit would be a block of a small number of consecutive

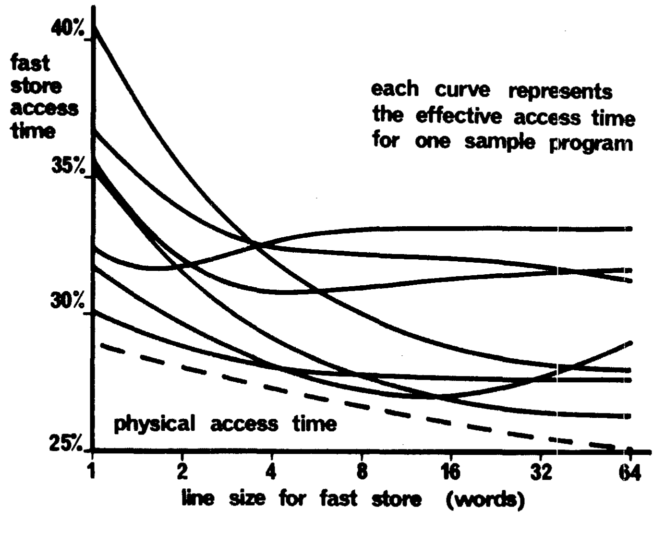


Figure 3—The physical access time of a fast store (broken line) and the effective access times (contnuouslines) for several sample programs. Access time is a percentage oı main store access time line size is in words, and store size is 1024 words

words accessed simultaneously from the main store. Such a block is very similar to, though much smaller than, a page in a paging system and will be called a line. Figure 3 shows how the effective access time of an integrated circuit fast store varied with line size during the simulation of seven sample programs. It can be seen that, when the line size is small, increasing it not only improves the physical access time of the fast store but also reduces the failure rate, resulting in an impressive performance improvement. But for larger line sizes any further improvement in the access time from the increased line size is offset by increasing failure rates and overall performance deteriorates. It appears that a line size of between four words and sixteen words is suitable, providing that the line size is not allowed to exceed the total width of the main store.

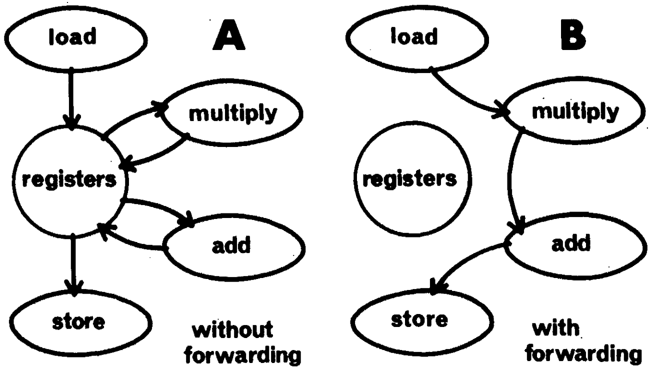
The associative fast store technique provides very hast effective access times and overcomes this problem in the design of very fast computers. Thus the onus is placed back onto the processor to make full use of the speed of the fast store, both by the provision of fast arithmetic units and by the execution of lengthy arithmetic operations in parallel. A beautiful technique for overlapping arithmetic operations has been developed by R. M. Tomasulo for the IBM 360/91 and is known as Tomasulo's algorithm.

Consider, for instance, the typical tight loop containing floating point load, multiply, add, and store instructions operating on the same register. As shown in Figure 4a the conventional machine places the result of each operation in the register before extracting it again to perform the next operation. The register has no substantial function in this loop which would be more efficiently performed as shown in Figure 4b. Here the partial result is passed directly from one arithmetic unit to the next without first being placed in the register, a technique known as forwarding. Not only is this faster, but it also frees the register from interlocks, which would prevent its concurrent use for subsequent calculations. Thus for the example loop, it might be possible to launch the second iteration of the loop before the first iteration has been completed.

The basic structure of a floating point arithmetic unit using Tomasulo's algorithm is shown, slightly simplified, in Figure 5. Separate arithmetic units are provided for addition and multiplication, and there are also units to hold the floating point registers and to buffer operands to be written to store. The arithmetic units are pipelines so that several independent operations, in different stages of completion, can be processed simultaneously within each arithmetic unit. Thus, for instance, the addition unit can start a further addition operation each logic cycle even though the individual addition operation takes three to four cycles to complete.

In front of each arithmetic unit there is a block of registers in pairs, the reservation stations. These serve to gather the operands required for the arithmetic operations as and when they become available. As soon as a reservation station has collected both the required operands, the relevant arithmetic operation can be started at this, the earliest possible, moment. Operands are made available to the reservation stations as early as possible by the cross bar switch which connects the outputs of all the arithmetic units, the registers and the store buffers to the inputs of all the reservation stations, so that any operand can be routed directly to any reservation station where it is required.

Tomasulo's algorithm applies only to operations between registers. Consequently arithmetic operations that derive one of their operands from store are per-



Figures 4a and 4b—The use of forwarding to speed arithmetic calculations

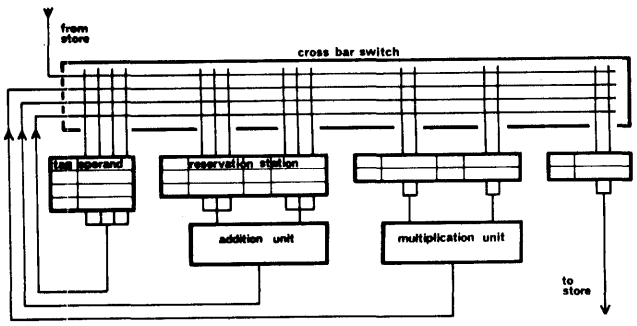


Figure 5—Typical floating point unit for use with Tomasulo's algorithm

formed in two stages, the first of which loads the operand from store into one of the store buffer registers while the second is a register to register operation between that buffer register and the specified floating point register.

Under Tomasulo's algorithm instructions are still decoded sequentially but their execution proceeds as and when the required operands become available. Arithmetic operations between registers are performed in four stages:

select a suitable vacant reservation station,
obtain both operands and place them in the reservation station,
execute the arithmetic operation,
transmit the result directly to all registers and reservation stations waiting for it.

The identity of the destination register must not be held with the operation as it is being processed, for arithmetic operations can be performed out of sequence and the result of some subsequent operation may already have been placed in that register. The essence of Tomasulo's algorithm is that a record is kept, for each register, of the origin of the result for which it is waiting, the result most recently assigned to it. Previous results, directed by the program to pass through the register, will be forwarded directly to the relevant arithmetic units and can be ignored by the register. The same technique is used for reservation stations, recording for each which operand or result it is waiting for.

As an example of the required effect, consider the short loop referred to above. The first instruction loads an operand from store to a floating point register. Obtaining the operand from store will take a small interval of time, even with an integrated circuit fast store, and so a store buffer is allocated, the register is set to wait for an operand originating at this store buffer, and the next instruction is considered.

This calls for a multiplication, and a reservation station in front of the multiplication arithmetic unit is allocated. One of the operands of the multiplication is being fetched from store, and one half of the reservation station waits for an operand from the store buffer allocated to this operation. The other operand is that to be loaded into the register as a result of the previous instruction, but this operand has not yet arrived at the register. Thus the other half of the reservation station is set to await this operand directly from the store buffer allocated to the first instruction, bypassing the register and making the operand available to the arithmetic unit at the earliest possible moment. The result of the multiplication instruction is to be placed in the register, which is set to wait for it. That the register will now ignore the operand from store is of no significance, for that operand will be routed directly to the multiplication unit, the only place where it is required.

The third instruction is an addition instruction, and an addition unit reservation station is allocated. Here too, one of the required operands comes from store and the other from a register which has not yet received the required result. In this case the register awaits a result from the multiplication unit and so the reservation is set to wait for an operand from a third store buffer and for the result of the multiplication. The register now awaits the result of the addition. The last instruction stores the contents of the register, causing one of the reservation stations for buffering operands to be stored to wait for the result for which the register waits, the result of the addition operation.

Meanwhile the two initial operands being fetched from store have probably arrived and been recognised by the multiplication unit reservation station. This has enabled the multiplication operation to start. The third operand fetched from store will be collected by the addition unit reservation station which will have to wait for the end of the multiplication for its other operand. In due course the result of the addition will be routed directly to the storage unit reservation station and also to the floating point register unless, as is likely, that register is already involved in subsequent operations.

The success of this scheme is entirely dependent on the registers and reservation stations being able to recognize the operands they require. Tag fields, shown in Figure 5 attached to each register and reservation station, are used for this purpose, containing a four or five bit tag identifying the origin of the required result or operand. Every operand presented to the cross bar switch is accompanied by an identification of its origin and each reservation station compares this with the origin of the operand for which it waits, so that the required operand can be recognised and acquired.

The origin will be one of the floating point registers, one of the store buffers or an arithmetic unit. For this purpose the arithmetic unit identification requires further elaboration since several arithmetic operations may be pending. Thus the origin defines the particular reservation station of the arithmetic unit rather than just the arithmetic unit.

The tag fields of floating point registers are readily set, for the register has been explicitly defined as the destination by an instruction which has already selected a store buffer or a reservation station. Thus the origin of the required operand is known. The register may have had its tag field set, indicating that it was still waiting for the result of some previous instruction. This tag field may now be overwritten with the new identification since that result will be routed directly to any reservation station needing it and there is no further need of it in this register.

Similarly if the operand required for a reservation station is to be obtained from a store buffer or is already held by its floating point register, then the identity of the origin of the operand can readily be inserted into the tag field of the reservation station.

But in many cases the required operand will not yet have reached the floating point register. In this case the identity of the register cannot be placed in the tag field of the reservation station, for the current contents of the register are irrelevant and out-of-sequence instruction execution may cause some subsequent result to have been already placed in the register before the required operand becomes available. However the tag field associated with the register indicates the origin of the operand for which it waits and which is required by the reservation station. Thus if the tag field of the reservation station is set to this value, then the required operand can be acquired by the reservation station directly it is presented to the cross bar switch and without ever passing through the register at all.

Tomasulo's algorithm is very effective and enables us to discount the time taken to perform floating point addition and multiplication, within limits. A machine equipped with a fast store and Tomasulo's algorithm is potentially capable of instruction rates approaching one per logic cycle. However the speed of the machine is reduced by the need to move data between the fast store and slower stores, conditional branches, interlocks on indices, and occasional very slow arithmetic operations such as division. In practice the machine can execute most programs at about one instruction per two logic cycles. Using current state of the art logic elements and high density interconnection techniques, logic cycles of 25 to 35 nanoseconds can be achieved, yielding an effective machine speed of 15 to 20 million instructions per second.

Both the associative fast store and Tomasulo's algorithm are standard well understood techniques and the machine described above is typical of large scale general purpose computers currently being developed for delivery in the next few years. A clear requirement exists for a small number of very much more powerful machines. These are required to perform lengthy repetitive "number crunching" in scientific applications such as weather forecasting and nuclear physics. There appears to be almost no limit to the useful speed of such machines, but the applications need a performance at least an order of magnitude greater than can be readily obtained with the techniques described above.

At first sight the attainment of a major improvement in performance is difficult to envisage. The fast store is already the fastest possible storage medium; we have used the maximum possible amount of logic in the arithmetic units, obtaining speed by brute force; the subtlety and complication of the control logic already approaches the limits beyond which it is no longer possible to detect and correct design faults and component failures; while the speeds of the logic elements and the sheer quantities used are such that the most important influence on the clock rate is probably the finite velocity of light.

However, detailed examination of the way in which the machine actually operated reveals that much of this potential power is squandered. The fast store, at least 128 bits wide and possibly 256, is capable of producing several operands every logic cycle; on average it is required to produce one operand every two logic cycles. Behind the fast store is the main store, a thin film store with a cycle time of perhaps 200 nanoseconds. Though its access time is long, this store will be well multiplexed and is probably capable of producing operands at the same rate as the fast store; it is rarely used. The floating point arithmetic units are capable of executing perhaps three instructions every logic cycle; they are normally required to execute one instruction about every three logic cycles.

All this power is wasted because the control logic is not able to decode instructions faster than one per logic cycle. Attempts have been made to design control logic capable of decoding several separate instructions simultaneously, but such control units are very large and hideously complicated, indeed so large and complicated that the speed of the machine may actually be reduced. The difficulty in the design of such units is the nominally sequential nature, and therefore the possible interaction, of the instructions being decoded in parallel. If these instructions were designed to be executed in parallel and were thus known not to interact, the problems of the control logic would be greatly simplified. Even better would be to forego the use of instructions designed to meet a general purpose se-

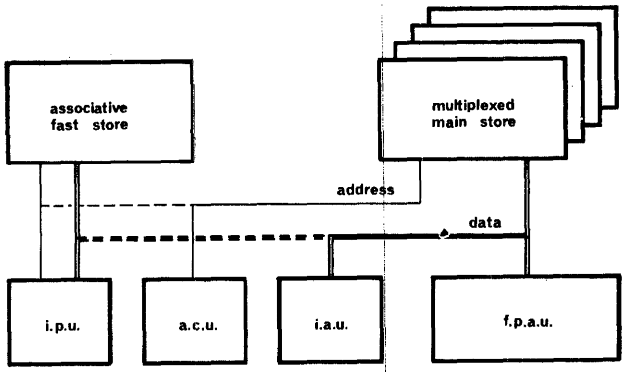


Figure 6—Typical hardware for a future large scale computer

quential programming concept, and to use instead instructions functionally oriented around the actual hardware present in the machine.

Figure 6 shows the hardware that might typically be present in a large scale general purpose central processor of the near future. The machine contains four main functional subunits:

an instruction prefetch unit,
an address calculation unit,
an integer arithmetic unit,
a floating point arithmetic unit.

There are two storage subsystems, a very fast integrated circuit store, with associative addressing, and a main store of longer access time but multiplexed to achieve a similar data rate. The stores are assumed to be 128 bits wide which, in view of the simulations referred to above, is the narrowest width likely to be chosen for a conventional machine and is also the narrowest width for which these proposals are feasible.

The Direct Functional Control scheme provides instructions to explicitly control this hardware so as to extract maximum performance for scientific calculations. This must be achieved with the minimum of extra hardware, less because of cost than because substantial extra hardware would be likely to lengthen the logic cycle and reduce the speed of the machine. Each of these instructions, which will have to be at least 128 bits long, provide control over all four major functional units for one logic cycle and one cycle only. But during that one cycle the programmer can control directly the arithmetic operations performed by each unit and the gating on their data buses.

The nature of such instructions is most readily understood by reference to the floating point unit which is shown in Figure 7. Comparing this with

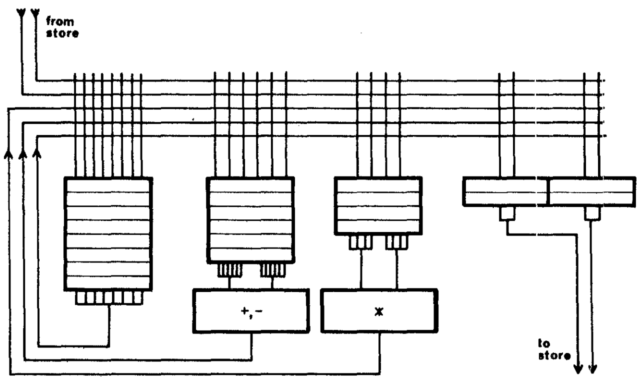


Figure 7—Detail of the floating point arithmetic unit

Figure 4, it can be seen that the hardware is substantially the same as for a conventional large scale computer employing Tomasulo's algorithm. Data is accessed from and returned to the store over double word highways so as to utilise fully the available width of the stores, the number of floating point registers has been increased to eight, and the gating between the reservation stations and the arithmetic units has been changed to enable any data item to be used as either operand of an arithmetic operation.

Figure 8 shows how the floating point unit is controlled by a part of the 128 bit wide instruction. This instruction contains fields to control the routing of data through the cross bar switch, the gating of operands between the reservation stations and the arithmetic units, and the arithmetic operations performed. The instruction controls all the operations of the machine during one logic cycle and one cycle only. Thus when an instruction selects the registers to be gated into the addition and multiplication units, units which

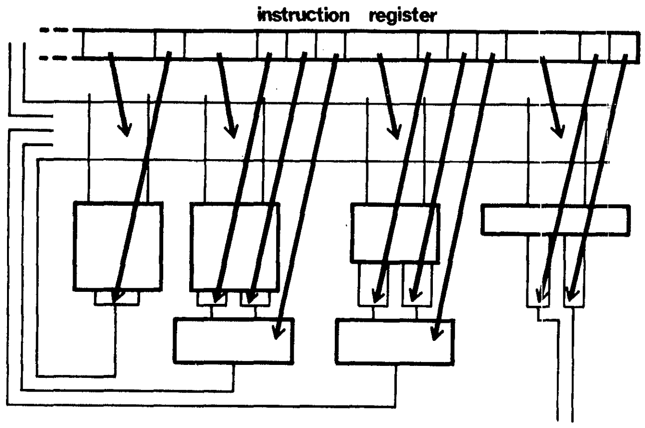


Figure 8—Control for the floating point arithmetic unit

take several cycles to perform their function, the results of these operations are of no interest to the instruction that initiated them. Instead the results of other addition, multiplication and store access operations, initiated by previous instructions, will be presented to the cross bar switch during this logic cycle. The instruction controls the switch to route these operands to their destination registers, and any result not routed to a register is lost.

Because of the very high speed of the design and because of its emphasis on the processing of large arrays of data, it is not advantageous to place array operands in the fast store. The high processing speed reduces the interval between delays due to accessing slower stores, and the large arrays force other useful information out of the fast store without themselves gaining any benefit. Consequently array operands must be fetched directly from the main store. Because of the multiplexing there is no reduction in the available data rate, but the long access time of this store forces the programmer to introduce foresight into his program, foresight that is readily available for array processing. The programmer is also expected to make use of the 128 bit width of the store to obtain his operands from the store in pairs.

By not using the fast store for operand access and by separating its store bus from that to the main store, it is possible to use this store for instruction fetchs and it can now produce one instruction 128 bits wide, every logic cycle. The address of this instruction is generated by the instruction prefetch unit, and may sequentially follow the address of the previous instruction, or may be a branch to another part of the program. A conventional machine would queue instructions in the prefetch unit until it was ready to decode them one at a time. In the proposed design the instructions, all 128 bits of them, are immediately decoded and executed without (in most cases) regard for queuing or interlocks.

The address generation unit forms the addresses of the operands needed from main store. The conventional machine would first search the fast store for these operands before accessing the main store. Now the fast store is fully occupied with supplying instructions and the operand addresses are despatched directly to the main store. The access time of the main store must be known to the programmer who must base his program on the generation of an operand address this predetermined number of instructions ahead of his requirement for the operand. He must also make allowances for the multiplexed nature of the main store so as to avoid clashing.

The integer arithmetic unit is principally required to increment the indices used for array accesses and to count round loops.

Control of the integer arithmetic unit, the address generation unit, and the instruction prefetch unit requires about 12 bits for each unit, out of the 128 available in the instruction. A 12 bit displacement field must also be provided for address generation. Frequently no branch is taken, and in such cases the 12 bits controlling the instruction prefetch unit are redundant. The addition of a single bit to indicate this enables these 12 bits to be used as a constant by the integer arithmetic unit or as an extention of the displacement field to 24 bits. Thus control of the 'integer half' of the machine can be obtained for 49 bits. In practice 16 integer registers may be found unduly restricting, and increasing the number of registers to 32 together with the provision of an additional loop counting facility, independent of the integer arithmetic unit, would require another 11 bits making a total of 60.

Control of the foating point unit consists of two parts: control of the individual arithmetic units and control of the cross bar switch. Control of the arithmetic units requires 20 bits to select the registers to be gated into the arithmetic units and to select the function for units that can perform several (for instance, addition and subtraction). The cross bar switch as illustrated contains 110 cross points, too large a number for direct control to be provided by a 128 bit instruction. However by restricting operations of limited usefulness, for instance gating two operands simultaneously into the same register, and by then adopting a simple coding system without loss of generality, it is possible to reduce the number of bits required to control the cross bar to 48.

The quantity of extra hardware required to provide direct functional control over the machine is quite small. Thus it should be possible to adapt a conventional design to operate in this mode without destroying its capability as a strict sequential processor using Tomasulo's algorithm and a fast store.

The principle implementation problem of the design is that the whole machine appears completely synchronous to the programmer. This causes difficulty when the machine has to be stopped, as the whole of a rather large quantity of logic has to be stopped at the same moment in time. The need to stop might occur because of an external interrupt, the absence of an instruction from the fast store or of a data item from the main store, interference in the main store due to peripheral transfers, or because of branching.

For most branching operations there is no need to stop the machine during the short interval of time before the new instructions become available. In many cases, for instance the end of the row of a matrix, it is possible to anticipate the branch and to continue to issue a controlled number of instructions from the previous instruction stream pending the arrival of the new instruction stream from the fast store. But events such as the detection of an exceptional case of an operand may necessitate stopping the processor while further instructions are fetched, to avoid erroneous further processing or the destruction of the operand.

The ability of the proposed design to issue a complex instruction every logic cycle and to make intensive use of the available hardware resources, results in a very powerful computing machine, a machine that is particularly attractive because its hardware cost is only marginally greater than for an ordinary large scale computer. This performance must be paid for chiefly in programming difficulty. Indeed if direct functional control was the only mode of operation of the machine, it would have to be abandoned as unprogrammable. However the machine can still be programmed with conventional sequential orders in exactly the same way as any other machine, direct functional control being used only for inner loops. The success of the design depends on the bulk of the code of a program being executed comparatively infrequently and on the inner loops in which the bulk of the processing is performed being comparatively small. It is then possible to make a reasonable decision as to the extent to which direct functional control should be used in any given problem.

The only other technique currently available which can yield computers of comparable power is the processor array concept developed by D. L. Slotnick and currently being implemented as ILLIAC IV. The direct function control scheme proposed here is substantially cheaper up to the limits of its performance, can tackle a wider range of problems, and because of its single instruction stream, single data set structure, is hopefully easier to program. Programming an application for the ILLIAC IV requires understanding of the problem so that it can be reformulated to fit the processor array, while direct functional control requires only local rearrangement of the code. But direct functional control can provide only a limited performance from a single processor, and a processor array can ultimately achieve a very much greater processing capability.

## REFERENCES

1 M J FLYNN
*Very high-speed computing systems*
Proc IEEE Vol 54 No 13 Dec 1966

2 C J CONTI D H GIBSON S H PITKOWSKY
*Structural aspects of the system/360 model 85 general organisation*
IBM Systems Journal Vol 7 No 1 1968

3 J S LIPTAY
*Structural aspects of the System/360 model 85, the cache*
IBM Systems Journal Vol 7 No 1 1968

4 R M TOMASULO
*An efficient algorithm for the automatic exploitation of multiple execution units*
IBM Journal of Research and Development Vol 11 No 1 Jan 1967

5 G H BARNES et al
*The Illiac IV computer*
IEEE Trans on Computers C-17 No 8 Aug 1968

6 D J KUCK
*Illiac IV software and application programming*
IEEE Trans on Computers C-17 No 8 Aug 1968

# A display processor design

*by* R. W. WATSON

*Shell Development Company*
Emeryville, California

T. H. MYER

*Bolt Beranek and Newman, Incorporated*
Cambridge, Massachusetts

I. E. SUTHERLAND

*Evans and Sutherland Computer Corporation**
Salt Lake City, Utah

and

M. K. VOSBURY

*Sanders Associates, Incorporated*
Nashua, New Hampshire

## INTRODUCTION

This paper describes the results of a collaborative design effort aimed at development of a general purpose display system for the SDS-940 time-shared computer.† The important features of the system evolved gradually from a number of separate design goals. We wanted a display system that would:

1. Contain an extensive but straightforward set of display generating commands.
2. Be able to generate pictures from highly complex data structures.
3. Allow easy access to display files from user programs in the main computer.
4. Provide some immediate feedback and interactive processing service to the display user, and be able to call upon the main computer for more extensive service.
5. Permit attachment of special purpose display generation and interactive hardware, as well as multiple display consoles.
6. Be capable of time-sharing its central resources among separate console-users.

These goals and their influence on the system design provide a framework for the detailed discussion that follows in the body of this paper. Before proceeding, however, we would like to give some orientation by presenting an overview of the system design without dwelling on our motives.

Figure 1 is a block diagram of the system. As indicated, its main components are a display processor (including computer interface) that controls the system and channels digital information among the other components, a display generator that produces ap-

* Formerly consultant to Bolt, Beranek and Newman, Inc.

† The system to be described was implemented by Sanders Associates, Inc. for Shell Development Company. An identical system was originally planned for delivery to Bolt Beranek and Newman Inc., but BBN's decision to replace their SDS-940 with a larger machine altered these plans.

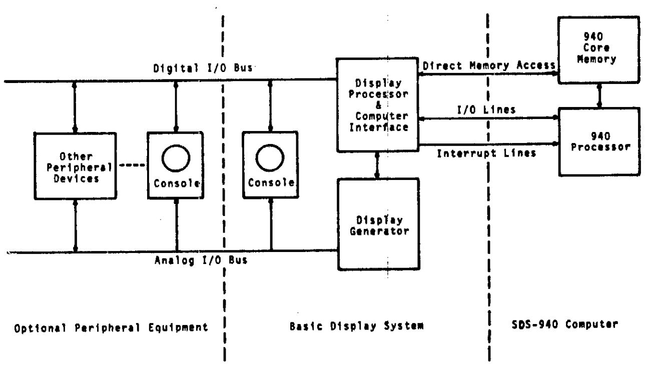


Figure 1—System configuration

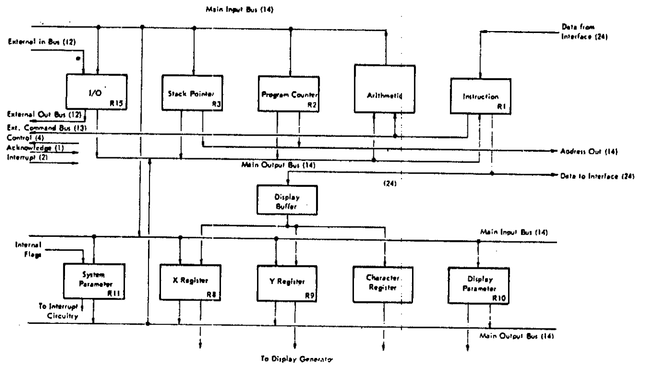


Figure 2—Processor

propriate analog drive signals, and a collection of display consoles and other peripheral devices Note that the display shares the memory of the central computer.

The display generator contains high speed vector, character and beam positioning generators. Display generators are discussed in references 1, 5, and 8, and the characteristics of the display generator for this system are discussed in reference 10. This paper is primarily concerned with the design of the display processor, and with certain aspects of the overall system design.

Figure 2 shows the display processor in more detail. One can view it as a collection of registers, each of which is connected to two main information paths—the Main Input and Main Output Busses. Other information paths provide connection to the display generator, peripheral devices, and computer interface.

Figures 3a – d describe the command set for the display processor. All commands are "immediate" in the sense that each contains its operand(s) in what is usually the address field (referred to in this paper as the "operand field"). The Display Commands (Figure 3a) supply information to the display generator via the X, Y, and character registers. The Address, Data and Miscellaneous Commands (Figures 3b – 3d) affect the contents of the various display processor registers, and may also cause information to be stored in 940 memory. Most of these commands contain separate fields to specify the operation and the register to be operated on. Because the various registers serve distinct functions, the effect of a given command will vary depending on the register specified. For example, a Load of the Program Counter is equivalent to a conventional jump; a Loading of the I/O register will have an entirely different effect. The I/O commands (Figure 3d) transfer digital data and control information to and from the peripheral devices, either directly, or via the I/O register.

As shown in Figure 1, the Display Processor and Generator, taken together, control and supply information to the peripheral devices via the Analog and Digital I/O Busses. Three paths allow the 940 computer, in turn, to control and inform the Display Processor. These are a direct connection between the

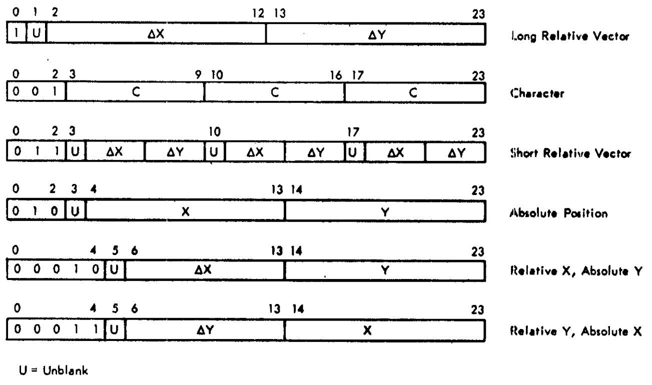


Figure 3a—Display commands

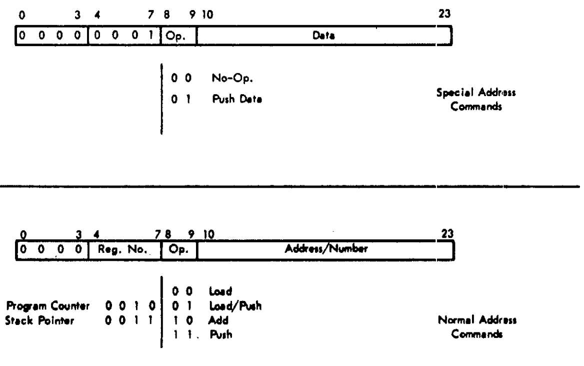


Figure 3b—Address commands

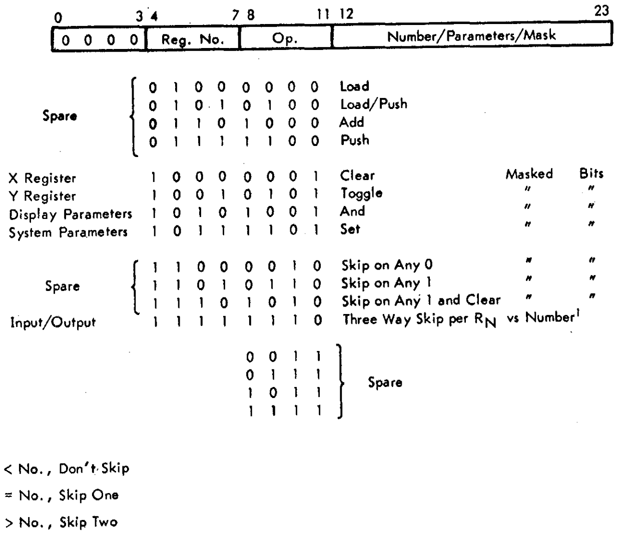

0 3 4 7 8 11 12 23
0 0 0 0 | Reg. No. | Op. | Number/Parameters/Mask
Spare
0 1 0 0 0 0 0 0 Load
0 1 0 1 0 1 0 0 Load/Push
0 1 1 0 1 0 0 0 Add
0 1 1 1 1 1 0 0 Push
X Register 1 0 0 0 0 0 0 1 Clear Masked Bits
Y Register 1 0 0 1 0 1 0 1 Toggle " "
Display Parameters 1 0 1 0 1 0 0 1 And " "
System Parameters 1 0 1 1 1 1 0 1 Set " "
Spare
1 1 0 0 0 0 1 0 Skip on Any 0 " "
1 1 0 1 0 1 1 0 Skip on Any 1 " "
1 1 1 0 1 0 1 0 Skip on Any 1 and Clear " "
Input/Output 1 1 1 1 1 1 1 0 Three Way Skip per $R_N$ vs Number[1]
0 0 1 1
0 1 1 1
1 0 1 1
1 1 1 1
Spare
[1] $R_N$ < No., Don't Skip
$R_N$ = No., Skip One
$R_N$ > No., Skip Two

Figure 3c—Data commands

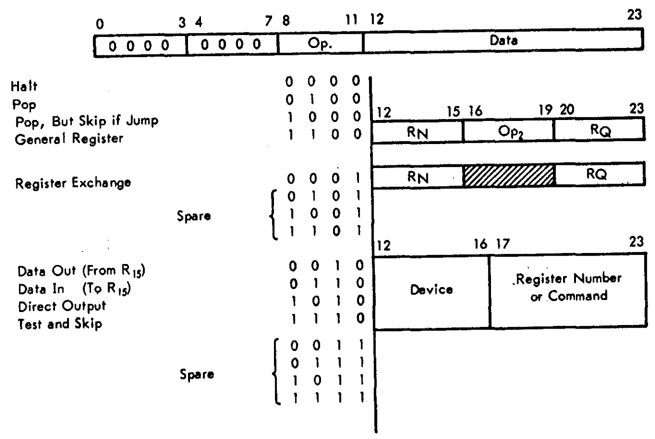


Figure 3d—Miscellaneous and I/O commands

Display Processor and 940 Core Memory, and two connections, the I/O and interrupt lines, between the Display and the 940 Processor.

*Display commands*

One of our goals was to design a rich but "clean" series of display commands. In particular, we wanted to avoid a difficulty we encountered in several other display systems—the fact that word length restrictions force reliance on two word instructions or on dual operating modes in which the machine will treat all words either as display data or instructions, depending on its mode. The 24 bit word length of the 940 provided enough space (just barely) to allow all instructions to carry OPCODE and X-Y or character data in a single word.

Figure 3a shows the six Display Commands. The display generator can produce lines and characters. Lines are drawn in $2 + 3L$ microseconds, where L is the length of the line in inches. The beam can be randomly positioned anywhere on the screen in 7 microseconds maximum. Characters are drawn in from 4 to 12 microseconds, depending on size and number of strokes required. One command plots three characters in "typewriter" format*; the remaining commands specify the endpoints of displayed lines.** The endpoint of a line can be specified, in two's complement, as an absolute location on the 1024 by 1024 coordinate grid of the display screen or as a relative displacement from the current beam location. One pair of commands allows endpoints to be specified in relative or absolute terms. Another pair allows mixed specifications—one coordinate absolute, the other relative. The remaining command allows three endpoints to be specified as short, relative displacements. Each X or Y component of a short displacement specification (Figure 3a) is represented, in two's complement, by one sign bit and two magnitude bits. The two magnitude bits are treated by the hardware as the two high order bits of a three bit magnitude representation. The low order bit is assumed to be 0. This allows displacements of about 0.1 inch in X and Y to be specified. Each line specification carries an unblank bit (U). If set, the line will appear, otherwise it will produce an invisible beam movement.

The appearance of displayed elements is controlled by the three fields of the display parameter registe (R1O), (Figure 4). Eight intensity levels and four character sizes are available. A line can be drawn solid, in a variety of dotted and dashed formats, or as a single dot at its terminal point (point plotting). To allow independent control of the three parameters, a masking mechanism is included.† To change parameters one uses a Load Command (Figure 3c) with bits 12–14 specifying which parameters are to be affected.

* A null code can be placed in the unused character position when it is desired to plot one or two characters. In addition, the character generator has an unusually rich complement of control characters, including space and half space up, down, backwards, and, forwards. Full details are covered in reference 10.

** The strting point for a line or group of characters is the, current beam position. The X and Y registers always contain this value; their contents are appropriately updated as each Display Command is executed.

† A similar scheme was used in the Digital Equipment Corporation (DEC) 340 and 338.

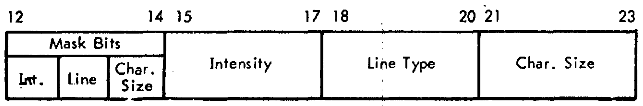


Figure 4—Operand field for display parameter command

*The pushdown stack*

One of our key goals was to achieve a display system that would allow us to represent pictures by means of complex data structures. Behind this goal was a desire to eliminate or minimize the separation that is necessary in many systems between a "master representation" and a "display file". Looking at this rather general goal in more detail, we wanted the ability to:

1. Execute nested picture subroutines to arbitrary depth.
2. Create "transparent" subroutines—save and restore selected display registers such as the X and Y beam position and display parameters on entering and leaving a subroutine.
3. Pass parameters to subroutines.
4. Easily identify objects selected by light pen or stylus in terms of the picture structure.
5. Perform certain forms of general list processing.

Nested subroutines can be handled by a variety of subroutine mechanisms. The need for easy light pen selection led us to use a pushdown stack. When processing a light pen or stylus "hit" one must trace one's path back through the subroutine hierarchy in order to relate the object selected to the drawing structure. Without a pushdown stack this requires search through the subroutine structure. With a stack system, however, the required trace is maintained compactly and automatically by the return addresses stored in the stack.

The use of a pushdown stack is not in itself new with this design. The DEC 338 display, for example, made very successful use of a stack system. What is unique in this display is the way in which the stack was implemented. The need to save and restore information other than return addresses meant that it had to be possible to push any register into the stack. In order to get the information back into the right register, data in the stack had to be marked in some way. After considering several marking schemes, we hit on the idea of placing *instructions* rather than data in the stack. When a display register is "pushed" into the stack, what actually appears in memory is an instruction to reload the register in question with its original contents.

The notion of putting instructions in the stack, of course, changes one's conception of the whole stack mechanism. The POP instruction (counterpart to PUSH), for example, becomes a special variety of "execute", and the stack pointer a kind of auxiliary program counter. In recognition of this, we reversed the direction in which stacks usually build. As information is pushed into the stack, the stack pointer is decremented. This means that instructions in the stack are "popped" (executed) in the usual low-to-high address order.

Treating the stack pointer as an auxiliary program counter suggested that we make it accessible, as is the program counter, to certain processor instructions. By doing so, we freed the stack from a fixed location in core. Because one can load the stack pointer, one is free to start the stack where one pleases. Moreover, as we shall see below, one can even achieve a stack that occupies disjoint areas of memory by saving the old stack pointer at the beginning of each new section of stack.

With this background, we can now look at some details of the stack system. The Push, Load/Push, and Push Data commands (Figures 3b and 3c) place information in the stack, Pop and Pop but Skip if Jump (Figure 3d) get it back out. As mentioned above, the Push commands assemble instructions in memory; the Pop commands execute these instructions. The Push operation may seem complex, but is in fact quite simple. To see this, let us examine a Push command in detail.

1. Assume "push the X Register" has been fetched into the Instruction Register (R1).
2. The register field (bits 4–7) of R1 selects the X register (R8). The contents of R8 are copied to bits 12–23 of R1.
3. Bits 8 and 9 of R1 are cleared to 0. The remainder of R1 is left unchanged.
4. R1 is copied back into the memory at the location selected by the Stack Pointer (R3).
5. The Stack Pointer is decremented.
6. The net result in memory is a "Load the X Register" command* with the current X value in its operand field.

The main use for Push is to save register contents for later restoration at the end of a subroutine. As indicated in Figures 3a and 3b, Push can be brought to bear on any register accessible to the programmer. Because the stack is marked, a single instruction restores the information regardless of where it came from.

* A variant of Push will place an Add Command in the stack.

In dealing with display structures, it is convenient to supply names or tags for the objects being presented. These may, for example, be pointers to other areas of memory that describe non-graphic properties of the objects. The No-Op command (Figure 3b) allows names to be included in a display file. It causes no action, but its operand field may contain tag information. Push Data allows names to be pushed into the stack, a further convenience when tracing back through a subroutine hierarchy. This command writes its own operand field into the stack in the form of a No-Op command.

The third Push variant—Load/Push—exchanges its operand field with the selected register before writing the original register contents into the stack. Load/Push the Program Counter provides a standard subroutine call. The current program location is stored in the stack (as a Jump instruction) while the Program Counter is simultaneously reset to the subroutine entry point specified by the Load/Push command. Load/Push can be used in a similar way to save and simultaneously reset any other register.

Load/Push the Stack Pointer deserves special attention. Because the Stack Pointer is loaded with the new value before its original contents are pushed, the old value will be pushed into the new stack Thus, the first word put into the new stack is a pointer that links it to the old stack. It is this feature that allows one to create disjoint stacks; the saved stack pointers provide an automatic address chain back to the original stack. We have chosen to call these stored links "Stack Jumps."

Pop, the counterpart to Push, causes the display processor to execute instructions in the stack. When the processor encounters a Pop, it increments the Stack pointer, fetches the instruction selected by the new pointer value, executes that instruction, and then returns to normal instruction execution under control of the Program Counter. Typically, the instructions executed by Pop will be Load or No-Op commands created by one of the Push instructions. However, any instruction can be executed through Pop.

With the Pop instruction in hand, we can now examine a typical subroutine linkage. Having entered the subroutine through a Load/Push Program Counter, one can use Push or Load/Push commands to save any other registers. The net result is a series of Load Commands in the stack with a Load Program Counter occupying the last (highest numbered) address. Two commands: Pop followed by a Jump to the previous instruction will restore the saved registers and provide a subroutine return. The processor loops on these two commands, reloading the saved registers, until the stored Load Program Counter removes it from the loop and returns control to the main program.

The Pop but Skip on Jump command allows one to restore saved registers *without* returning from a subroutine. This command behaves exactly like Pop except upon encountering a Load Program Counter in the stack. In this event the stacked instruction is ignored, the Stack Pointer decremented and the Program Counter incremented an extra time. The net result is that the processor breaks out of a loop such as the one suggested above, just before executing the return Jump.

The above discussion has suggested some conventional uses for the stack instructions. However, such features as the ability to manipulate the Stack Pointer in various ways permits the user to devise more sophisticated uses for the stack mechanism. We have made heavy use of this flexibility in the software support package. One example application is the handling of rubber band lines and other simple constraints within the display processor. We accomplish these functions by performing list processing in the display file using the stack feature.[11]

Experience in working with the system has shown that the heavy use of multiple stacks could be more efficient if another stack pointer were available or if a 14 bit address length general purpose register were available for temporary storage of the Stack Pointer. The Shell system is being modified to add two such 14 bit general registers. The ability to execute instructions in the stack has given generality and power to the display processor at modest cost.

## *Memory sharing*

A consequence of our desire to achieve close coupling between pictorial and other information was the need to allow easy access to display files from programs in the 940. As well as permitting advanced graphics applications, we felt that close access would simplify the general software support for the display.

To realize this goal we attached the display processor directly to the core memory of the central computer rather than relying on a separate buffer memory.* The display processor addresses the 1.75 microsecond 940 memory through its program counter and stack pointer. In operation, the display processor refreshes the display consoles by executing display commands stored in 940 memory and passing the data they contain to the display generator.

Given this close interconnection between display

* This connection utilizes the 940's second memory port.[4]

and main computer, considerable care was necessary to ensure a display system that could operate effectively without degrading or endangering the supporting time-shared computer system. One potential danger—competition between display and central processors for memory access—was reduced to an acceptable level by use of dual access priorities on the second path to memory.**[4]

A second and more serious danger—inadvertent alteration of 940 memory by a display program—was eliminated by including memory mapping and protection hardware in the display processor. This equipment is identical in function to equivalent hardware in the 940.[4] By means of this mapping, the 16K word "virtual" memory that can be accessed by the display (and 940) instructions is mapped into 2K word physical pages that may be scattered through the 64K words of 940 core memory. At any one time only a few of these pages may be assigned to the display, and those pages that are assigned may be made accessible for reading only or for reading and writing.

Registers in the mapping hardware indicate, for each of the eight pages that the display might address, whether or not a physical page is assigned, and if assigned its status (read only or read/write). Only the 940 monitor can change the contents of the map registers. As shown in Figure 5, memory addresses transmitted by the display processor, are processed through the mapping hardware before accessing 940 memory. Any attempt to address an unassigned page or to write into a read-only page stops the display processor and sends an interrupt signal to the 940.

One consequence of mapping is that undebugged display programs are of no danger to the system or to other users. Mapping has the additional benefit of allowing users and system software designers to treat display programs in exactly the same way as 940 user programs. In fact, because mapping for a user's 940 program can be made identical to the mapping for his display file, the two can share the same

**The 940 CPU accesses memory through the first path to memory. The display accesses memory through a second path. Devices on the second path can request access with either higher or lower priority than the first path. The display processor overlaps the drawing of a vector or character with the fetch of the next command. Memory accesses at this time are with low priority. When the display operation is completed, access is made with high priority, if not previously successful. Non-overlapped accesses are made with high priority. Using the above mechanism, reasonable assumptions on command mix and the fact that the 940 memory has 4 independent interleaved modules, it has been estimated that the 940 CPU will be blocked from immediate memory access less than 2 percent of the time.[10]

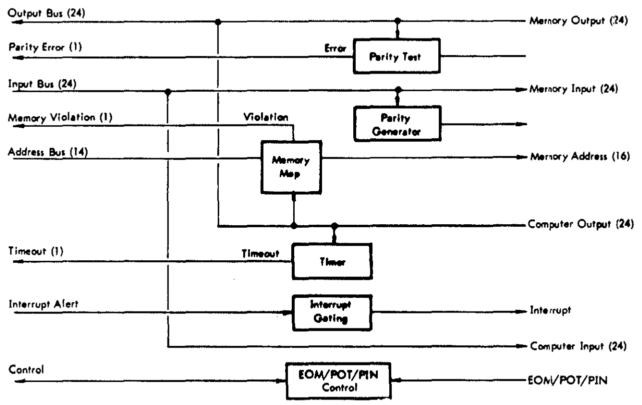


Figure 5—Interface structure

address space and thus, be merged in any way the user pleases. Thus, the user can, if he wishes, create a common data structure that represents pictorial and other properties of the objects to be viewed. In addition, he can achieve an unprecedented richness of interaction between operations performed at a display console and the underlying processing in the main computer.

## *Processing tasks—Display vs. 940*

The issue of how much power to include in the display processor is a complicated one. This issue is discussed more fully in an earlier paper that was inspired by the difficulties we encountered on his project. We chose to include enough computing power to handle the immediate response to interactive events such as light pen "hits" or the depression of push buttons. Less than this would yield sluggish interaction; tasks requiring more power could, we felt, be relegated to the 940 processor.

With these ideas in mind, we equipped the display processor with a set of commands aimed specifically at interactive situations. As shown in Figure 3c, these include bit manipulating and skip commands and an arithmetic compare operation. The bit manipulating and skip instructions include Clear, Toggle (Complement), And, Set, Skip on 0, Skip on 1, Skip on 1 and Clear, all handled under the mask in the operand field of the instruction. These commands are used to test or change status, control interrupt masking and so forth. There is also a three way arithmetic compare of a selected register with the operand giving a skip of 0, 1, or 2, depending on the result. This command allows one to branch on the X or Y location of the display beam or of a coordinate input device. Taken together with the Add, Register Exchange and General

Register Commands,† and the stack mechanism, these interactive commands have allowed us to do such things as handle light buttons, produce point rasters, and perform the work involved in light pen tracking, all without intervention from the 940. Control of the display processor is implemented with microcoding and a read-only memory. The time required per microstep is 400 nanoseconds. Command fetch, decoding, and program counter update require 6 microsteps plus a memory read time. The number of microsteps required per command execution is variable, Load requires 1, Push 3 and Pop 9, for example. The Pop and General Register Commands have the longest execution time. The read-only memory can be easily modified or inexpensively replaced. This feature will be used to modify or add commands thought to be useful from the software experience.[11]

In spite of its power, the display processor must call on the 940 for assistance in tasks beyond its capabilities. In addition, the 940 must, of course, have ultimate control over the display. We satisfied both needs by connecting the display processor to the I/O and interrupt systems of the 940. Through these connections the display processor can transmit service requests to the 940. The 940 processor can in turn interrogate and set the registers of the display. Together with the shared memory mechanism, these two connections yield a closeness of coupling that contributes importantly to the ability of the two machines to share their processing resources.

Through its I/O lines the 940 processor can directly access all registers of the display. Any display register can be brought into the 940 processor by a 940 Parallel Input (PIN) instruction. Conversely, the 940 processor can set any display register through a Parallel Output (POT) instruction. This feature aids the 940 in initializing the display and in processing interrupt requests. If the 940 sets the display's Instruction Register (through a POT instruction), the display will treat the information as a command, execute it, and then halt. Unless directly altered by a command executed in this way, the display's Program Counter is not changed. The net result is that the 940 can, in effect, "execute" any display instruction. As well as access to the display registers, the direct I/O connection allows the 940 to stop and start the display set the display's memory map and the "device map" described in the next section.

†Though not directed at any particular interactive function, our implementation of the processor design allowed us to include these commands at little cost. They have proven more than worth the price. The Add (Figures 3b, 3c) and Register Exchange (Figure 3d) generates a new processor instruction in which $OP_2$ operates on $R_N$ using the contents of $R_Q$ as operand. This allows one, for example, to add or compare two registers.

The interrupt system gives the display a means for requesting help from the 940. Some events in the display (irrecoverable errors, for example) can only be dealt with by the 940. Either the 940 or the display processor can cope with other situations (light pen hits, scope edge violations). In recognition of this, we grouped all interrupt as well as other control and status information into one register—the System Parameter Register (R11), shown in detail in Table I. The bottom twelve bits of this register are accessible both to the bit manipulation commands of the display and, via the POT/PIN instructions, to the 940. The top seven bits are accessible only to the POT/PIN instructions because only the 940 can deal with the information they contain.

TABLE I—System parameter register*

| *Bit* | *Function* |
|---|---|
| | (Bits Accessible to 940 Only) |
| 5<br>6 | These two bits assist the 940 in interpreting certain interrupt events. |
| 7 | Parity Error Flag. |
| 8 | Memory Map Violation Flag. |
| 9 | Time-Out Flag (the display has a built-in down-counting clock). |
| 10 | Halt Mask. |
| 11 | Halt Flag. |
| | (Bits Accessible to 940 or Display) |
| 12 | Unused. |
| 13 | X Edge Overflow Flag. |
| 14 | Y Edge Overflow Flag. |
| 15 | Edge Overflow Mask. |
| 16 | Synchronous Hit Flag (e.g., light pen). |
| 17 | Synchronous Hit Mask. |
| 18 | Asynchronous Hit Flag (e.g., pushbutton or keyboard). |
| 19 | Asynchronous Hit Mask. |
| 20 | Blink (toggles continuously at blink rate). |
| 21 | Blink Control. |
| 22 | Slow Mode Control (for storage tube consoles). |
| 23 | Master Unblank (if 0 unconditionally blanks the display). |

* Nineteen of the possible 24 bits in this register were implemented.

The lower bits in R11 handle several kinds of events, for each of which there is a flag bit and a mask bit. The flag bit is set whenever the event occurs; the setting of the mask bit determines whether or not an interrupt signal is sent to the 940. This arrangement allows the programmer to cope with events through the bit oriented instructions of the display processor, or ignoring them in his display program, to pass them on as interrupt signals to the 940. In addition, a display program can request service from the 940 by executing a Halt and Interrupt instruction (Figure 3d).

Because the 940 must assist the display processor in certain situations, it was necessary to allow display users to write real-time 940 programs. The problem of preventing real-time programs from degrading the time sharing performance of the 940 was handled by setting limits on a display user's CPU usage during each refresh cycle of the display.

### *Consoles and other I/O devices*

So far, we have considered the display processor and its relationship to the parent computer. We were also concerned with display consoles and other peripheral devices, and their relationship, in turn, to the display processor and generator. Our main goal in this area was flexibility. We wanted the ability to attach a variety of display consoles, differing in some cases in their equipment complements, as well as other non-display devices including graphic input tablets, and specialized analog equipment, such as circle or raster generators. We met this need by dissociating from the display processor design any consideration of individual consoles or other devices. Instead, we elected to treat these as I/O devices, and to handle their control and the transmission of information to and from them by means of a very general I/O bus system.

The digital portion of this bus system is similar in nature to the bussing schemes used on several general purpose computers. Devices are selected by an address field in the I/O instructions; all devices are treated homogeneously as collections of registers; and a given register may contain control or status information, input or output data, or a mixture of these.

Figure 3d shows the Input/Output commands. Two of these permit the user to transmit information between the I/O register (R15) and the registers of external devices. Incoming data and status information can then be examined by the Display Processor, through the test and skip instructions described in the last section, or dealt with by the 940 through the POT/PIN commands. The remaining two commands permit somewhat faster direct output of key commands and direct testing of key device status bits. As mentioned in the last section, another component in the digital I/O bus system is the channeling, through OR gates, of synchronous and asynchronous events in the peripheral devices into the H1 and H2 bits of the System Parameter Register.

Corresponding to this treatment of digital information, the transmission of analog signals within the system was also handled through a bussing scheme, which allows input of analog signals to summing points within the display generator as well as output of display drive signals.* Because of this treatment of peripheral devices, one can view the display processor and generator taken together as a specialized hybrid computer whose main job is to handle a series of I/O devices through a combined analog/digital bus system.

Just as the 940 processor is time-shared, we wanted the ability to time-share the display processor and generator among a number of user consoles without danger of interference between them. This was achieved by giving the 940 processor the ability to control and thus schedule, usage of the display processor, and by allowing for device protection hardware in the display's I/O bus design. This hardware utilizes a mapping scheme similar to the memory mapping and protection hardware in the 940 and has the additional advantage of allowing a user to refer to peripheral devices through "virtual" addresses that can remain constant even though he may be assigned a different console at different times.

## CONCLUSION

The stack mechanism in this design is the most significant departure from previous machine design practice. The features of a marked stack, and the ability to create disjoint stacks (through the "stack-jump" linkage) are both easy to implement and useful. As is by now well known, the stack feature in a display processor is essential for orderly treatment of "hits" detected by the light pen or other stylus devices.

Close coupling between display information and 940 programs has been achieved by the mechanism of shared memory. Other general purpose display systems seem to be relying more and more on small local computers for interactive service and to shield the main computer from the display. By contrast, we deliberately set out to achieve a rich interaction between display and parent computer, and the extremely close coupling

* Whether a device generates or responds to analog signals depends upon bit settings in its control register.

of the two machines reflects this goal. Our experience so far indicates that this coupling can be achieved without serious degradation of the 940 time-sharing system.

Until now most displays have been treated strictly as I/O equipment. As displays have grown in complexity over the years, however, we have come to recognize that display processors have many of the attributes of general purpose computers. In recognition of this, we deliberately approached the design problem with a processor-oriented rather than I/O device-oriented approach. This thinking is reflected in the display's extensive instruction set, in the use of memory and device mapping, in the uniform treatment of consoles as peripheral devices, and finally, in the microcoding and uniform bussing scheme that dominate the display processor design.

## ACKNOWLEDGMENT

The authors would like to acknowledge the contributions of their co-workers at Shell Development, Bolt Beranek and Newman, and Sanders Associates, during the various design and implementation phases of this project. The Bolt Beranek and Newman effort was supported by the Advanced Research Projects Agency of the Defense Department under contract F 19628–68–C–0125.

## REFERENCES

1 L C HOBBS
*Display application and technology*
Proc IEEE 59 12 1870-1884 1966

2 N A BALL H Q FOSTER W H LONG I E SUTHERLAND R L WIGINGTON
*A shared memory computer display system*
IEEE Trans on Electronic Computers EC-15 5 750-756 1966

3 K H KONKLE
*An analog comparator as a pseudo-light pen for computer displays*
IEEE Trans on Computers C-17 1 54-55 1968

4 W W LICHTENBERGER M W PIRTLE
*A facility for experimentation in man-machine interaction*
AFIPS Proc 27 589-598 1965

5 C MACHOVER
*Graphic CRT terminals-characteristics of commercially available equipment*
AFIPS Proc 31 149-160 1967

6 T H MYER I E SUTHERLAND
*On the design of display processors*
Com ACM 11 6 410-414 1968

7 M W PIRTLE
*Intercommunication of processors and memory*
AFIPS Proc 31 621-634 1967

8 H H POOLE
*Fundamentals of display systems*
Spartan Books Washington D C 1966

9 C SEITZ G F PFISTER
*A display processor for a small computer*
AFIPS Proc This issue 1969

10 R W WATSON
*The design of a general purpose graphic terminal for a time-sharing system*
Shell Development Co Technical Progress Report 138-68 July 1968

11 R W WATSON et al
Paper in preparation describing the design philosophy of the software for use with the display system reported here.

# The system logic and usage recorder

*by* R. W. MURPHY

*International Business Machines Corporation*
Poughkeepsie, New York

## INTRODUCTION

A fundamental problem in monitoring the performance of a system with a hardware device is too much data. Inside the System/360 Model 40, for example, seventeen address bits and sixteen data bits may be processed every 2.5 microseconds; this rate is equivalent in bulk to about three novels per second but not generally equivalent in interest or information. The design objective for any hardware monitor, therefore, is to reduce the data it sees as soon as possible.

The associative memory (AM) is an excellent means for not recording data beyond significance. The memory can be instructed to record data only if they are new; if the data have already been seen and stored, no more space need be squandered upon them. This philosophy of monitoring and measurement has been expanded into the System Logic and Usage Recorder, an experimental device under test in IBM Poughkeepsie's SDD Advanced Technology group.

In the Recorder, the basic associative processes of interrogation and storage are extended, by means of a system of data routing and field control, into a capability for performing advanced data reduction and data processing algorithms. The algorithms are programmed and retained in a control storage where they may be added to or modified by the user.

Data to be analyzed in the Recorder are collected at the host computer through a special monitor interface which detects and transmits such signals as instruction and data addresses, operation codes, and the statuses of channels and internal computer conditions. The monitor interface, which consists of 48 lines, is one-way, and does not affect the operation of the host computer. In addition to the monitor interface, there is a standard input/output interface which is used to pre-load the associative memory when this is required by an algorithm, and over which the collected and reduced data are transmitted as the Recorder's output.

In this paper, some simple data-gathering procedures are discussed first in order to introduce the design concepts of the Recorder. This is followed by descriptions of the organization and programming of the system, and finally some specific data reduction algorithms are given.

### *Simple data gathering and basic operation*

A question asked in performance measurement is, "How much time is spent in executing programs out of various areas of storage?" To determine these times, a counter must be assigned to each of the active areas; when an instruction is fetched from an area, clock pulses begin incrementing the corresponding counter, and continue until an instruction is brought from some different area.

In the Recorder, the counters are assigned to storage areas automatically, through associative memory. Initially the memory is blank and the counters stand at zero; but when the first instruction address is received in the Recorder from the computer being monitored, it is stored in an associative memory word cell as shown in Figure 1.

This word cell then becomes responsible for monitoring the storage area 00100 through 001FF, which the word cell does by comparing its contents with each new instruction address brought into the AM input register. As long as there is equality in the high-order

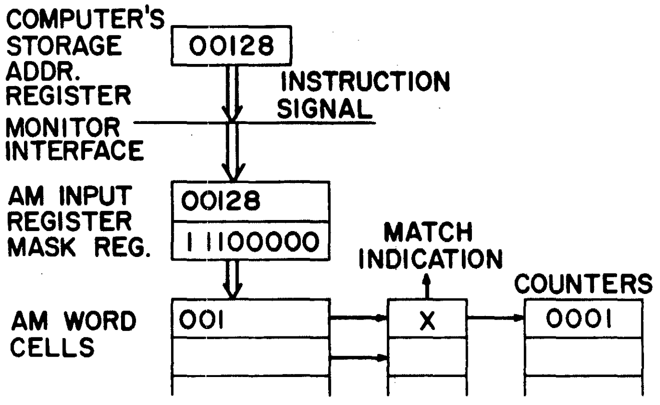


Figure 1—Assignment of counter to initial execution area

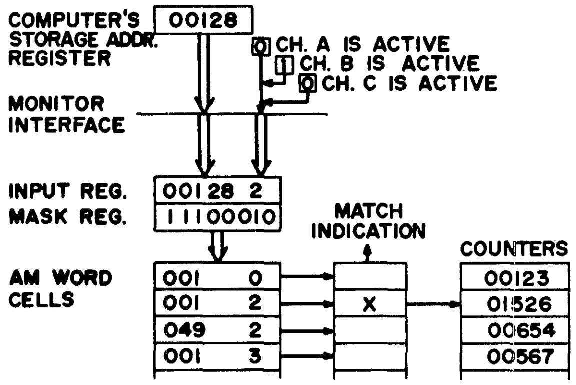


Figure 3—Correlation of executed area with channel activity

bits of the address (the low-order bits are ignored by means of a mask), a match will be indicated, and the match indicator for that cell will continue the selection of the corresponding counter, allowing it to accumulate time intervals.

This process of interrogation is repeated until an inequality between the value stored in the cell and an instruction address produced a mismatch, signalling that program execution has moved to a different area of monitored storage. The mismatch will deselect the counter, and will cause the controlling program to branch into a write cycle in order to record a new active area as shown in Figure 2.

The process diagrammed in the figure will assign counters as they are needed, and record their assignment in the associative word cells. Since interrogation of the associative memory is a single operation, it does not matter how many of the cells contain meaningful data, and the fineness of the measurements can be

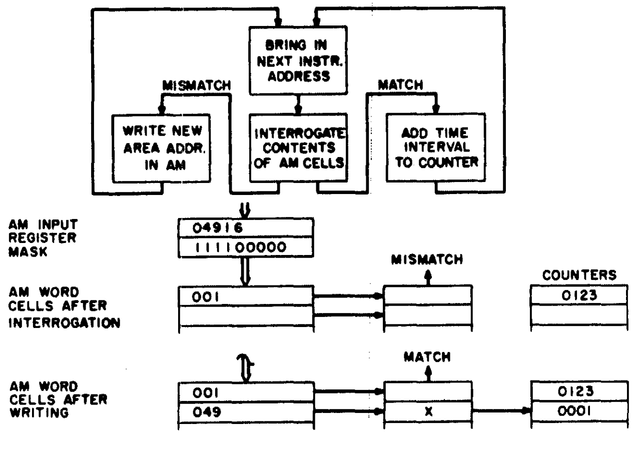

BRING IN NEXT INSTR. ADDRESS
MISMATCH
MATCH
WRITE NEW AREA ADDR. IN AM
INTERROGATE CONTENTS OF AM CELLS
ADD TIME INTERVAL TO COUNTER
AM INPUT REGISTER MASK
04916
11100000
MISMATCH
COUNTERS
AM WORD CELLS AFTER INTERROGATION
001
0123
MATCH
AM WORD CELLS AFTER WRITING
001
049
X
0123
0001

Figure 2—Assignment of next counter to next execution area

adjusted by means of masking to take advantage of the available memory space. If execution in the host computer should revert to an area already identified by the Recorder, such as 001 in the example, the original cell's contents will again match the address and reactivate the counter for additional accumulations.

The two-branched monitoring procedure is a basic one, and can be made to yield many kinds of information. For example, if channel activity is also monitored and presented at the interface as a field of bits, this field can be juxtaposed with the instruction address field as in Figure 3.

With this process, which has the same flow chart as in Figure 2, a correlation will be made automatically between storage usage and channel activity. It is, of course, immaterial what kind of data is being brought to the interface; the user can perform the correlation on any combinations of events which are represented by digital signals brought over the monitor interface.

Another form of correlation is of interest because it yields information about the sequence of events taking place in the monitored system. This procedure consists of relating each event to its predecessor by forming an ordered pair at the AM input register as in Figure 4.

Two kinds of events are recorded in this process: the occupancy of a particular area, and the transition from one area to another. The procedure is essentially the same as that given by the flow chart of Figure 2, except that an additional data routing is programmed. Each address is first placed into the left-hand field (the current field) and the interrogation is performed. Following the action consequent on the interrogation, the address is then put into the right-hand field (the previous field) and is retained there until the next address arrives and the cycle is repeated.

This procedure develops a graph of the system's

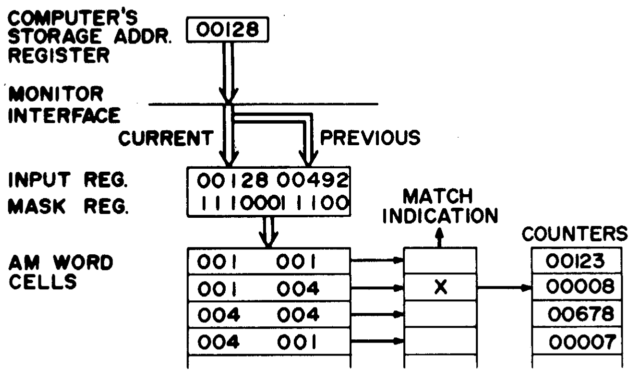


Figure 4—Recording occupancy and transitions of execution areas

operation in associative memory, and could be used to study the operation of paging algorithms. If the full instruction address were applied to the memory by modifying the mask, all the linkages of a program would be recorded and could be used to draw the program's block diagram as it was actually executed. The application would be very wasteful of space, however, and impractical except for very small programs. There is a more complex procedure, to be discussed later, which eliminates much of the redundant information and makes block diagramming feasible with associative memories that will be available in the near future.

Emphasis so far has been placed upon the associative operations and what might be called the logic recording capability. The usage recording functions take place in the counters, which are actually cells in a supplementary storage addressed by the associative memory as a result of interrogation operations. These cells may be set up in various ways to record counts, times, or the presence of computer conditions, according to the measurements required.

### *General design concepts*

The examples of data gathering just discussed show that a variety of performance measurements can be made, simply by changing the nature and the positioning of data applied to the associative memory. This variety is enhanced greatly by means of a stored program control system which gives the user full control over the functions available in the Recorder. In general, each step of a data reduction procedure will specify the following elements:

*Routing.* The source, length, and terminus of a field of data to be processed.

*Masking.* The suppression of part or all of a field at a particular step in the procedure.

*Operation.* Interrogate, store, or read for associative memory.

*Branching.* Choice of the next step, based upon results of previous steps.

The specification of these elements applies primarily to associative memory as it processes the data received from the monitor interface, and is incorporated in the AM format instruction:

| Oper. Code | Routing 1 | Routing 2 | Mask | Next Instr. 1 | Next Instr. 2 |
|---|---|---|---|---|---|
| | | | | | |

The operation code for the AM format instruction will specify one of the following:

INTERROGATE—compare contents of input register with all stored words and turn on match indicators for cells with equal contents.

INTERROGATE NEXT—same as above, except that the match indicator for the next cell is turned on.

WRITE—store the contents of the input register into all cells whose match indicators are on.

WRITE NEW—store the contents of the input register in the first vacant word cell.

WRITE ONE—store the contents of the input register in the first cell whose match indicator is on.

WRITE ALL—store the contents of the input register in all cells regardless of the match indicators.

READ—put the contents of the first cell whose match indicator is on into the output register.

Two fields of data may be moved simultaneously by means of the two routing specifications. These fields may be one, two, or three bytes in length, or, alternatively, a literal constant of one byte may be substituted for one of the routing specifications. The routing of data will be discussed in more detail in the section on Data Paths and Routing Control in conjunction with the data paths of the Recorder.

In general, the fields of data processed are of variable length, on a byte basis. The associative memory is eight bytes in width, and its masking is also generally controlled on a byte basis. However, many algorithms require status bits which must be masked or unmasked by bit. The mask specification in the instruction, therefore, consists of fifteen bits, of which the first seven apply to the first seven bytes of the associative memory, and the remaining eight to the individual bits of the eighth byte. In addition, it is also possible to apply a literal mask to any byte by placing it in a routing specification along with an identifying code. This literal mask has precedence over the normal mask, and remains until removed by another literal. This mask is not normally used in data reduction, but is necessary for such algorithms as simultaneous addition into associative memory or ordered retrieval from it.

The next two instruction specifications of each instruction provide conditional branching to the program, based upon the collective condition of the match indicators. The choice of the next instruction depends on the following:

| | | |
|---|---|---|
| INTERROGATE | —if single or multiple match | Instr. 1 |
| | if no match | Instr. 2 |
| WRITE or READ | —if one or more MI's are on | Instr. 1 |
| | if no MI's are on | Instr. 2 |

*Data paths and routing control*

Figure 5 is a schematic diagram of the data registers and paths of the system. Each line represents a path for one byte of data, and a dot where two lines cross indicates a programmable connection. One group of six paths (48 bits) carries monitored data from the interface with the host computer to the input of the associative memory. The various registers and the crossbar switch provide buffering and field control over these data. Another path, one byte wide, connects memory outputs to memory inputs through an adder to allow internal processing functions.

The word logic circuits link the supplementary storage with the associative memory and provide an addressing function for the two memories. This addressing function is initiated by interrogating the associative memory with data in its input register; if the data in any associative word cell compare equally with the interrogating data, either that word cell, or a word cell in supplementary storage in one-to-one correspondence with it, or both may be selected for the entry or recovery of data. Explicit addresses for these word cells do not appear in the instructional

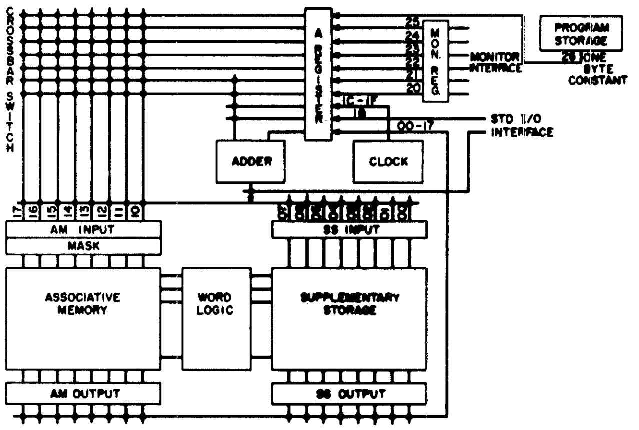


Figure 5—Recorder data paths

control system. The word logic circuits also provide other functions, including tie-breaking in the case of multiple matches and a match/no-match signal for conditional branching in the program.

Control over the data routing is accomplished within the instruction by means of routing specifications. The standard instruction format contains two routing specifications, each controlling one field of data; a special instruction format is used for supplementary storage operations which are to be overlapped with the associative operations. The routing specification in the standard format contains 16 bits, identified as follows:

*Change Code* (one bit). A zero indicates that the A Register is to be left unchanged; a one causes the specified field to be entered into the A Register before being routed further.

*Literal Code* (one bit). A one causes a one byte constant from the instruction to be entered into the A Register before being routed further. This constant replaces the field length and source address specification.

*Length Field* (three bits). Specifies the number of bytes of the field being routed. The maximum field length from the monitor register is three bytes, and from other sources, seven. A length of zero causes no transfer of data.

*Source Address* (six bits). Specifies the location at which the lowest-order byte of the field to be routed is to be found. Successive bytes the same field are moved in accordance with the length specification.

*Terminus Address* (five bits). Specifies the location to which the lowest-order byte of the field is to be routed. Addresses are tabulated below.

| *Sources* | | *Termini* | |
|---|---|---|---|
| Supp. Store Output | 00-0F | Supp. Store Input | 00-0F |
| Assoc. Mem. Output | 10-17 | Assoc. Mem. Input | 10-17 |
| Void | 1A | Void | 1A |
| I/O Input to Recorder | 1B | I/O Output from Recorder | 1B |
| Clock | 1C-1F | | |
| Monitor Interface | 20-25 | | |
| Constant | 26 | | |

*Notes*: Addresses are given in hexadecimal.

The address for the constant is not used when the constant is specified as a literal, but if the value of the constant is unchanged the constant may be routed either alone as a one-byte field, or as part of a two- or three-byte field at addresses 25 or 24.

If a void is specified as a source, the corresponding terminus is reset to zeros.

If a void is specified as a terminus, positions of the A Register corresponding to the source are reset to their new values.

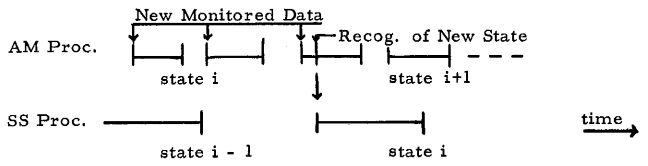
The two routing specifications per instruction permit two fields to be moved simultaneously and in parallel from the monitor interface to the associative memory input register via the A register and the crossbar switch. Transfers of data from sources other than the monitor register take place over a bus which is one byte wide, and are therefore serial by byte. As a result, only one such transfer can be called for in each instruction, using the first routing specification. The second routing specification can be used, however, for a simultaneous transfer through the crossbar. A literal can be specified only with the second specification.

*Supplementary storage*

Supplementary storage (SS) is used to retain times, counts, and condition codes for which associative processing is not required. However, each word cell of supplementary storage corresponds to a unique cell of associative memory and may be selected wheneevr an interrogation of associative memory turns on the match indicator for the corresponding AM cell. The general concept is that the AM cell retains data describing the state of the monitored machine, while the SS cell collects the statistics relative to that state.

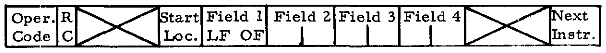
The character of the monitoring algorithms is that there is a series of operations involving associative memory only, establishing or identifying a record for the monitored machine's state. This process will usually be completed only when the computer has assumed a new state, but a match indicator will be on, pointing to the record of the previous state. If the algorithm provides an SS instruction at this time, the SS cell will be selected and updated according to the SS instruction. Once the selection has been made, it is not affected by any alteration of the match indicators until the SS instruction is completed and another one issued.

It may be seen from Figure 5 that the updating is accomplished through the adder and the SS input and output registers, and that it is possible for AM and SS operations to proceed independently once the selection of an SS cell has been made. This overlap will take place automatically for all AM instructions except those which call for the transfer of data between associative memory and supplementary storage or over the I/O channel. The overlapped processing may be represented as follows:



The time at which the monitored computer assumes a new state is taken to be the time of receipt of new monitored data, as indicated by the appropriate strobe signal from the computer. Since there is generally a lag of one cycle before the new state is recognized, the clock is buffered so that it may be reset to record a new time period starting from the strobe while the old time period is retained pending use in the SS instruction. If no new state has occurred, the old and new time periods are combined.

The updating of a word in supplementary storage is controlled by a single instruction containing specifications for performing different operations on four fields of the word. These fields may be from one to seven bytes in length individually, the combined length not exceeding the sixteen bytes of the SS word. The SS instruction occupies control storage as part of the programmed algorithm, but it differs in format from the AM instruction:



Notes: RC = reset controls
LF = length of field
OF = operation on field

The starting location specifies the low-order byte of field 1, which is updated according to its length and operation specification. The remaining fields are contiguous in the SS word, and are processed in succession. If the entire sixteen bytes of the word are not utilized in an application, the starting location may be other than zero, and the time of completion of the SS instruction will be lessened.

In addition to length, the field specification may call for one of the following operations:

1. Increment field
2. Add clock to field
3. Put the lesser of the clock reading and the old field value in field.
4. Put the greater of the clock reading and the old field value in field
5. OR the interface byte to the field
6. No operation

*Application examples*

In the application examples to follow, the algorithms are given as block diagrams, in which each block represents one instruction, including data routing, the operation, and the masking for AM operations. Data are routed by fields, which are constant within each application and are designated by capital letters generally mnemonic with their meaning. The location of a field is indicated by a subscript identifying the register involved in the routing or the memory itself. These subscripts are:

b—monitor interface buffer
a—crossbar entry register
i—associative memory input register
s—storage cells of associative memory
o—output register from associative memory
p—input/output registers of supplementary storage

The various fields used in an algorithm form an ordered set at the input to associative memory and after being written into a particular word cell. The notation for such an ordered set is:

$<S_sP_sC_s>$ for a particular stored word

If interrogation is to be performed, it is generally on a set of such words. This set is not ordered and is written as follows:

$\{<S_sP_s—>\}$

In this example, S and P identify the fields active in the interrogation, and the dash indicates that the field occupying that relative location in the word is masked.

*Application 1: Combinations of events and states*

**Problem**

To find out what system states occur over a period of operation of a host system, how many times each state occurs, and how much time is spent in each state. For this application, a system state is defined to be one combination within the following classes of monitored signals:

| | | | |
|---|---|---|---|
| Stopped/operating | 2 | possibilities | 1 bit |
| Running/waiting | 2 | " | 1 bit |
| Supervisor/problem | 2 | " | 1 bit |
| Channels busy | 8 | " | 3 bits |
| Page of instruction | 256 | " | 8 bits |

The monitor interface is set up to provide all of the above signals except page of instruction on an on-off basis. The page of instruction is the high-order 8-bit group of the instruction address, whose presence at the interface is signaled by means of the instruction strobe. An evaluation of the system state is to take place at each instruction strobe, or, if instructions are not being executed, at each change in the remaining conditions.

**Procedure**

Each system state is represented by a particular bit pattern in the above array of 14 bits, and is recorded in one word of associative memory. The time interval and usage of each state is totaled in the corresponding word of supplementary storage. If instructions are being executed (operating and running program states), the entire bit pattern is used, otherwise only program and channel statuses are stored.

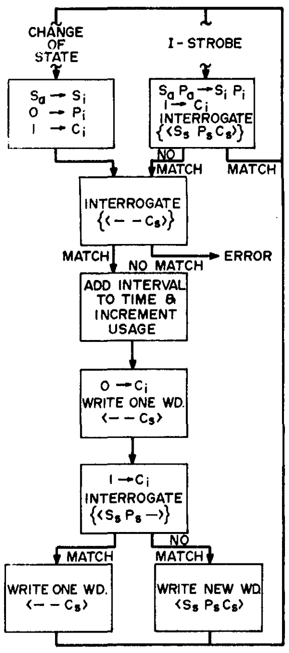
Whenever a change of state occurs, the appropriate bit pattern is compared simultaneously against all those previously stored. If no match if found, indicating a new state, the bit pattern is stored in the next vacant word, and the statistical fields in supplementary storage are initialized. If a match is found, indicating a repetition, the statistics are updated.

Interrogations of associative memory may occur as a result of instruction strobes without a change from the state of the previous interrogation. To detect changes, a control bit is added to the array of 14 bits and is set to one in the word representing the current state of the system.

### Results

At the end of the evaluation, there will be one word of data for each different system state which has actually occurred. These can then be printed out using the ordered retrieval procedure to present the non-executing states first, then the states in page order.

### Algorithm for combination of events and states


CHANGE OF STATE
I - STROBE
$S_a \rightarrow S_i$ $O \rightarrow P_i$ $I \rightarrow C_i$
$S_a P_a \rightarrow S_i P_i$ $I \rightarrow C_i$ INTERROGATE $\{\langle S_s P_s C_s \rangle\}$
NO MATCH
MATCH
INTERROGATE $\{\langle - - C_s \rangle\}$
MATCH
NO MATCH
ERROR
ADD INTERVAL TO TIME & INCREMENT USAGE
$O \rightarrow C_i$ WRITE ONE WD. $\langle - - C_s \rangle$
$I \rightarrow C_i$ INTERROGATE $\{\langle S_s P_s - \rangle\}$
MATCH
NO MATCH
WRITE ONE WD. $\langle - - C_s \rangle$
WRITE NEW WD $\langle S_s P_s C_s \rangle$

S - Field combining program status and busy channels bits (6 bits total)

P - Page of instruction (8 bits)

C - Last state indicator

Fields in Storage: $\{\langle S_s P_s C_s \rangle_{\text{assoc.}}$

$\langle \text{time} + \text{usage} \rangle_{\text{supp}}\}$

When an instruction strobe initiates a cycle, the monitored bits are routed through the A register to the I/E register for interrogation of associative memory. A match indicates no change of system state and completes the cycle.

If instructions are not being executed, the change of program or channel status starts the cycle, in which only S bits are taken from the Monitor buffer and zeros are put into the P field of the I/E register.

Before the new state is recorded, the time in the last state must be added to the total for that state and the usage incremented. This is accomplished by interrogating with the last state indicator in order to select the corresponding word in supplementary storage. Routing of this data through the adder is not detailed.

After resetting the last state indicator, an interrogation is made with the P and S fields still in the I/E register to determine if the current system state is one which has been previously recorded. If it has, only the last state indicator is stored in preparation for the next cycle; otherwise, the entire contents of the I/E register are written into the next vacant word to record the new state.

*Application 2: distributions of events*

### Problem

The path length between branches taken may be defined as the number of sequential instructions executed before a branch to a nonsequential address. It is important in determining how far a computer ought to look ahead in its instruction fetches. This application determines what path lengths actually occur in programs and how frequently each occurs. The distributions are to be found for paths preceding each type of branch instruction.

The signals which the monitor interface supplies are the operation code, a bit indicating whether the branch was taken, and an instruction strobe. If the bit for 'branch taken' were not available, then the address and length of instruction could be used to make an arithmetic check for nonsequential instructions.

### Procedure

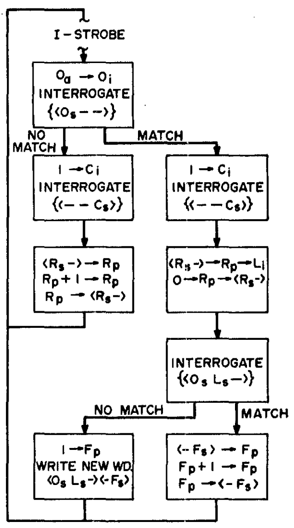
The associative memory is preloaded with the set of operation codes of the branch instructions, one word for each code. These words also contain a one in a single-bit field to indicate "branch taken," and a path length field containing zero. In addition, a specially marked word is set aside for the running count which is initially zero.

As each operation code and "branch taken" bit is brought in at the monitor interface, the combination of these two fields is compared against the preloaded set of codes. If no match is found, the running count field is incremented by one. If a match is found, the running count field is routed to the path length field, unmasked, and a second interrogation made. If this also results in a match, the frequency field of the matched word is incremented; otherwise, the new path length is stored in a vacant word with an initial frequency of one. The running count field is reset to zero.

### Results

At the end of the run, there will be one word stored for each path length and operation code. These might then be printed out using the range retrieval procedure to condense the different path lengths into groups.

### Algorithm for finding distributions of events


I-STROBE
$O_a \rightarrow O_i$ INTERROGATE $\{\langle O_s - -\rangle\}$
NO MATCH
MATCH
$1 \rightarrow C_i$ INTERROGATE $\{\langle - - C_s\rangle\}$
$1 \rightarrow C_i$ INTERROGATE $\{\langle - - C_s\rangle\}$
$\langle R_s -\rangle \rightarrow R_p$ $R_p + 1 \rightarrow R_p$ $R_p \rightarrow \langle R_s -\rangle$
$\langle R_s -\rangle \rightarrow R_p \rightarrow L_i$ $0 \rightarrow R_p \rightarrow \langle R_s -\rangle$
INTERROGATE $\{\langle O_s L_s -\rangle\}$
NO MATCH
MATCH
$1 \rightarrow F_p$ WRITE NEW WD $\langle O_s L_s -\rangle\langle - F_s\rangle$
$\langle - F_s\rangle \rightarrow F_p$ $F_p + 1 \rightarrow F_p$ $F_p \rightarrow \langle - F_s\rangle$

O - Operation code and "branch taken" bit (preloaded set of branch codes with L = 0)

L - Length of path

C - Code for running count word

R - Running count

F - Frequency

Fields in Storage - $\{\langle O_s L_s C_s\rangle$ assoc. $\langle R_s F_s\rangle$ supp$\}$

Each instruction strobe initiates a test to find if a branch was taken for one of the prespecified operation codes. These need not be the entire set of the host computer.

If no actual branch is found, the running count is incremented by selecting the word where it is stored with an interrogation for its code. The field is read out of supplementary storage, routed through the incrementer, and restored in the same word.

If the branch has taken place, the running count is routed into the I/E register where it becomes the path length. The combination of operation code and path length then is either stored, or if already in storage, causes an increment to be made to its frequency field.

*Application 3: Short sequences and mixes*

### Problem

Knowledge of instruction mixes can be an important factor in the planning of new systems. There are a number of ways in which the collection of mix data can be specified, all involving some form of sequence following or finding. In this example, the problem is to find what operation codes immediately precede the conditional branch types of instruction, up to a maximum of six including the branch.

### Procedure

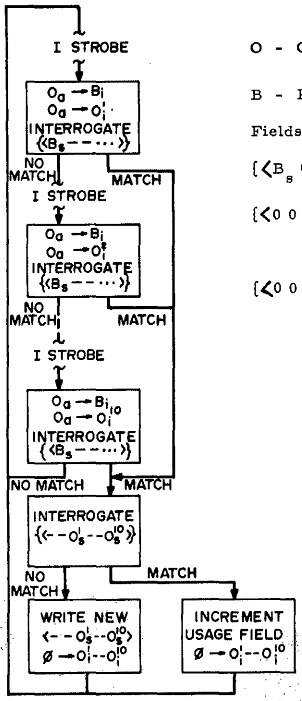
One word of associative memory is to be used fos each mix, with the operation codes distributed acrosr the word in six fields of one byte each. As operation codes arrive at the monitor interface they are routed to successive fields in the Interrogate/Entry register and also to a field set aside for comparison against the set of conditional branch codes which occupy a special set of preloaded words. When one of these codes is found, the array of six fields in the I/E register is used to interrogate the rest of associative memory which holds the arrays already found, and the appropriate entry or updating of usage is performed. The I/E register is reset to zeros, and the next operation code starts a new sequence.

The sequence may go beyond five codes before a conditional branch is found. In that case, the seventh code takes the place of the first, and so on until a conditional branch is found.

### Results

Each word contains one mix of six or fewer operation codes. The terminating conditional branch code may occupy any of the six fields, but if there is at least one zero after it, the entire sequence is as recorded; if not, the preceding five codes are read in "end-around" fashion.

### Algorithm for finding short sequences


I STROBE
$O_a \rightarrow B_i$ $O_a \rightarrow O_i^1$ INTERROGATE $\{\langle B_s - - \cdots\rangle\}$
NO MATCH
MATCH
I STROBE
$O_a \rightarrow B_i$ $O_a \rightarrow O_i^2$ INTERROGATE $\{\langle B_s - - \cdots\rangle\}$
NO MATCH
MATCH
I STROBE
$O_a \rightarrow B_{i_{10}}$ $O_a \rightarrow O_i^{10}$ INTERROGATE $\{\langle B_s - - \cdots\rangle\}$
NO MATCH
MATCH
INTERROGATE $\{\langle - - O_s^1 - - O_s^{10}\rangle\}$
NO MATCH
MATCH
WRITE NEW $\langle - - O_s^1 - - O_s^{10}\rangle$ $\emptyset \rightarrow O_i^1 - - O_i^{10}$
INCREMENT USAGE FIELD $\emptyset \rightarrow O_i^1 - - O_i^{10}$

O - Operation code received at monitor interface

B - Pre-stored branch operation codes

Fields in storage:

$\{\langle B_s\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\rangle\}$ set of branch codes

$\{\langle 0\ 0\ O_s^1 O_s^2 O_s^3 B_s^4\ 0\ 0\ 0\ 0\ 0\ 0\rangle\}$ mix of four codes in sequence of four

$\{\langle 0\ 0\ O_s^{11} O_s^{12} O_s^{13} O_s^{14} O_s^{15} B_s^{16} O_s^7 O_s^8 O_s^9 O_s^{10}\rangle\}$ mix of ten codes in sequence of at least sixteen

Successive operation codes are placed in successive O fields across the I/E register by means of a string of macroinstructions differing only in the routing microinstruction. When a branch operation code is received, a common routine is followed to add the new mix to storage or increment the usage field of an existing mix.

*Application 4: Long sequences*

### Problem

One way of determining the performance of a system is to see how often prespecified sequences of events occur. In this example an operating system is to be tested with a known load to determine if predicted sequences of supervisor calls, interrupts, and object programs are being followed. The sequences may be very long, may overlap or include each other, and may start or end with any arbitrary element.

The change to a new current PSW represents a step in the sequence, and can be detected by the fact that there is an interruption in the host system or that a LOAD PSW instruction is executed. The address of the PSW identifies the sequence element and is obtained from the monitor interface whenever a change occurs.

### Procedure

Associative memory is preloaded with the sequences to be followed, the elements of each sequence being placed in successive memory words. In the word the code for each element occupies one field, in this case 24 bits of address. The word also contains two single-bit fields, one of which contains a one for the start and the other a one to indicate the end element.

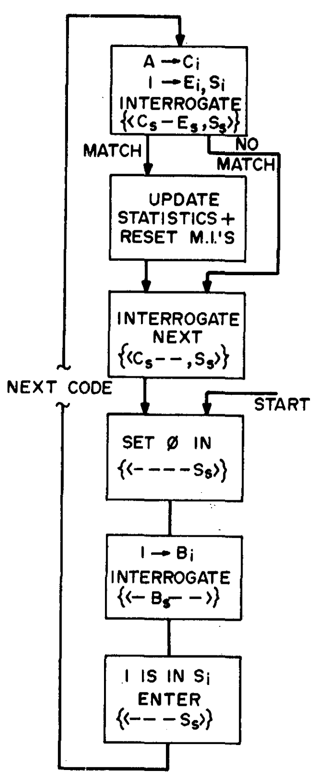
This procedure makes use of a special interrogation operation for associative memory in which, when a word is matched, the next succeeding word in physical order is selected for the entry of data. In this case, a status bit is entered after this form of interrogation in order to keep track of progress through the sequence, and the crucial interrogation is made simultaneously on the address and status bit. If the interrogation is successful after the next element has been received, the status bit is moved to the next word.

In addition to recording successes in traversing complete sequences, statistics can be compiled on partial traverses in the words of supplementary storage corresponding to intermediate sequence elements.

### Results

At the end of the test, associative memory will contain the sequences tested for, and supplementary storage the record of how well these sequences were followed. The sequences could then be printed as a complete test record in a format permitting an item by item comparison with results of tests of variations of load or system.

### Algorithm for following long sequences



| | |
|---|---|
| C - | Code for sequence element |
| B, E - | Start and end |
| S - | Status bit |

Initially, and at the end of each cycle, the status bits are set to one for all first elements.

When the next code is received from the monitor interface, an interrogation is first made to find out if that code matches any expected last elements of sequences so far successfully followed. If so, the statistics are updated and that element is reset to zero status (without affecting other elements in that sequence).

The same code then is used to interrogate the set of all elements whose status bit is one. This operation uses the INTER NEXT operation to prepare for the eventual entry of a one in the status bit of the next word. Figure 6 shows the match indicators turned on for the word actually matched.

Zeros are then set into all status bits, regardless of the match indicators, and without resetting them. This step clears any elements which may not have been matched with this last code.

Finally, all first elements are selected for entry by the use of a normal interrogate operation. This se-

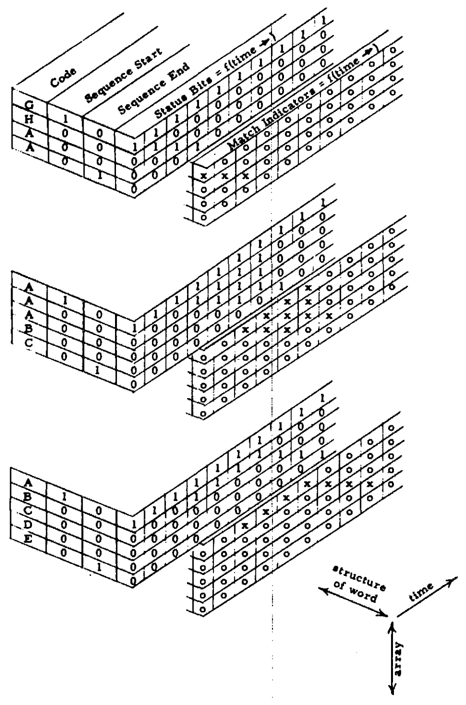


Figure 6—Steps in following the sequence GHAAAABCDE

lection is OR'd with the selection obtained by the INTER NEXT operation above, so that ones can now be entered into the union of the two sets.

*Application 5: Block diagramming*

### Problem

In debugging or in evaluating the performance of a program it is important to know whether program segments are executed in the proper order, how much time is spent in each segment, how well they were overlapped with channel activity, and if execution was forced to wait. Although one or a few segments might be singled out for examination by methods similar to those of the preceding applications, there is difficulty in predicting where and what to look for, and a chance of missing something significant.

If every instruction address were paired with its successor in the instruction stream and the combination applied to associative memory, eventually the memory would contain all the links between instructions for that program. However, most instructions have unique successors, and the technique would waste memory space or redundant information. The essential information is contained in just those linkages from or to instructions which have several successors or predecessors. These linkages can be identified from addresses and operation codes in the instruction stream.

### Procedure

Each word of associative memory contains three address fields, the "entry," "exit," and "destination." The entry and exit addresses are the first and last of a block of sequential instructions, and the destination is the entry of a succeeding block, so that each stored word represents one linkage in the logical structure of the program.

Certain addresses are identified as exits when they occur in the instruction stream accompanied by a branch operation code. The first address after an exit is automatically an entry to a current block, which will occupy one of four possible relationships to blocks already found. As the entry and succeeding addresses appear in the instruction stream, they are compared with previously stored entries and exits to resolve whether the current block is new or one being retraced, or whether either the current block or an old block is to be partitioned.

As execution of the program proceeds, with repetitions of its segments, most of the linkages will be followed one or more times, and the corresponding division of the address stream into blocks will be established. When these elements are found or repeated, their time and usage is noted, and channel and wait statuses are correlated with them, using supplementary storage for this additional data.

### Results

It can be shown that each conditional branch instruction will result in at least two, and no more than four linkages, and that the number of blocks established by the branch is always one less than the number of linkages. Since one word of storage is required for each linkage, approximately 2700 blocks can be recorded in a 4096-word memory. Depending upon the complexity of the program's structure, the memory can cope with programs of between 6,000 and 16,000 instructions.

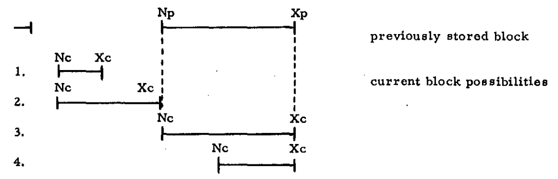
At the conclusion of a block diagramming evaluation, associative memory will contain the structural composition of the program according to its actual execution, and supplementary storage will contain the statistics correlated with each structural element. The standard presentation of this information would be a listing of the blocks with their exit linkages governing their order.

Once the information has been collected, other output procedures can be used to meet special requirements. For documentation of the program, it may be desirable to present the block diagram in pictorial form, using the host computer to compute and print the diagram. When the program is being optimized by trial, it will not always be necessary to print out the entire listing, but only the more time-consuming elements.

## Detailed description of procedure

If an instruction is a conditiona' branch, the first time its operation code is found in the instruction stream, it is recognized to have he potential for a different successor in some future execution and therefore it is recorded as the "exit" of a block. Its successor of the moment is one "destination" and also an "entry" to another, or possibly the same, block. The basic record thus consists of three addresses, identifying the entry, exit, and one destination of the block.

When a conditional branch identifies the next address as an entry to a block, this block may intersect some block already derived from the instruction stream. There are four possible relationships of a current block to blocks already traced out, as shown in this diagram



In the first possibility, none of the addresses from the current entry, Nc through the current exit, Xc, will be found to match any previously stored entries or exits, Np or Xp; the block is therefore new and can be added to the store.

A current entry may not be recognized, but may be followed eventually by an address which does match some previously stored entry. The address just previous to that matching Np becomes the current exit of a block, as shown in 2. above, and the block is recorded with Np as destination. The program will continue by repeating <Np Xp>, because Xc is not a branch.

The destinatiop of a block may be to an entry already recorded, as shown in case 3. Assuming that no change of operation code has taken place the same exit must follow, and the block need not be recorded again unless the destination is different. Eventually, in the program's execution only case 3 will be found.

If a branch, conditional or unconditional, has led to a new entry within a block, as shown in case 4, this fact will not be known immediately. However, sooner or later an address will match the exit, Np, to signal the condition. The current block can be added to the store, but the previous block is intersected by it.

In order to partition the intersected block discovered in this case, it is necessary to determine the address one location less than the current entry. This exit is not computable exactly when variable-length instructions are being executed, but it might occur again in the instruction stream and be recognized because its successor matches the entry in question. To cause this to occur, a flag is added to the intersected block, removing it from use by the algorithm, so that if the block should be repeated from its original entry, the situation will resolve itself into case 2.

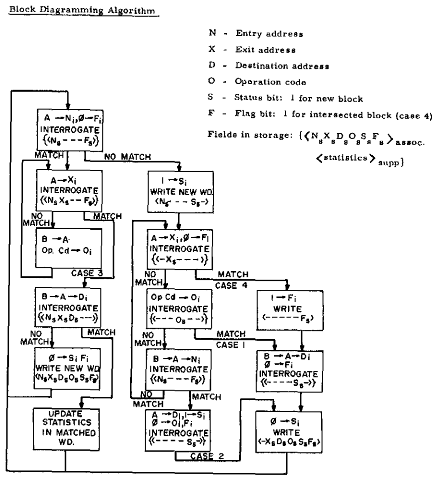
The flagged block might include an initializing routine which is never repeated, and the block will contain time and status data which cannot be distributed to its partitions. Therefore, the flagged block is retained for the ultimate readout and presentation of results

Special operations in the program, such as multiway branches, cause no difficulties to the operation of the algorithm when they are based on recognized operation codes. If the program changes an operation code to a branch, as mentioned in case 3 above, the algorithm must be altered to take into account some cases in addition to the four cases described. An algorithm which makes use of addresses only, and is thus unaffected by a changed operation code, has been worked out by the author but is not included here.

## ACKNOWLEDGMENTS

The author expresses his appreciation to Mr. R. R. Seeber for his advice and encouragement on this project.

The associative memory for the system logic and Usage Recorder was designed by Mr. A. W. Bidwell, and system design construction and debugging were performed by Messrs. F. E. Jordan, H. L. Wetzel, and G. T. Manelski.


Block Diagramming Algorithm
N - Entry address
X - Exit address
D - Destination address
O - Operation code
S - Status bit: 1 for new block
F - Flag bit: 1 for intersected block (case 4)
Fields in storage: {⟨$N_s X_s D_s O_s S_s F_s$⟩assoc. ⟨statistics⟩supp}
A → $N_i$, Ø → $F_i$ INTERROGATE {⟨$N_s$ - - - $F_s$⟩}
MATCH
NO MATCH
A → $X_i$ INTERROGATE {⟨$N_s X_s$ - - $F_s$⟩}
1 → $S_i$ WRITE NEW WD ⟨$N_s$ - - - $S_s$ -⟩
NO MATCH
MATCH
B → A Op. Cd → $O_i$
CASE 3
A → $X_i$, Ø → $F_i$ INTERROGATE {⟨- $X_s$ - - - -⟩}
NO MATCH
MATCH CASE 4
B → A → $D_i$ INTERROGATE {⟨$N_s X_s D_s$ - - -⟩}
Op Cd → $O_i$ INTERROGATE {⟨- - - $O_s$ - -⟩}
1 → $F_i$ WRITE ⟨- - - - - $F_s$⟩
NO MATCH
MATCH
NO MATCH
MATCH CASE 1
Ø → $S_i$ $F_i$ WRITE NEW WD ⟨$N_s X_s D_s O_s S_s F_s$⟩
B → A → $N_i$ INTERROGATE {⟨$N_s$ - - - $F_s$⟩}
B → A → $D_i$ Ø → $F_i$ INTERROGATE {⟨- - - - $S_s$ -⟩}
NO MATCH
MATCH
UPDATE STATISTICS IN MATCHED WD.
A → $D_i$, 1 → $S_i$ Ø → $O_i$, $F_i$ INTERROGATE {⟨- - - - $S_s$ -⟩}
Ø → $S_i$ WRITE ⟨- $X_s D_s O_s S_s F_s$⟩
CASE 2

# Implementation of the NASA modular computer with LSI functional characters

*by* J. J. PARISER

*Hughes Aircraft Company*
Fullerton, California

and

H. E. MAURER

*NASA Electronics Research Center*
Cambridge, Massachusetts

## INTRODUCTION

The NASA Electronics Research Center (ERC) in Cambridge, Massachusetts, has undertaken a broad program to satisfy flight computer system requirements for future missions, including versatility and long term reliability. Specific attention to these requirements is necessary because flight qualified aerospace computers and even some still under development, have been designed for increased computational speed and arithmetic capability, but not for the long life reliability and application flexibility that will be required for future space missions.[1,2] For example, the mean time between failure (MTBF) of available aerospace computers lies in the range of 2,000 to 5,000 hours, whereas long space missions will require an MTBF of $10^5$ hours.

Sseveral computer organizations have been described in the literature which include redundancy for increasing mission reliability, but still neglect applications versatility.[3,4] Some non-spaceborne computers of the array or multiprocessor type are currently being developed.[5,6] These systems, although potentially capable of meeting ERC's versatility and reliability objectives, lack design features for space applications (component reliability, weight, volume, radiation hardness, etc.).

This paper describes the architecture of a modular computer which can be configured to operate as a number of parallel processors, with each segment or column solving an independent problem that may be different or identical. Each column in turn contains a number of blocks called modules, which may be configured so as to form patched columns, using modules from different physical locations; for example, a diagonal (see Figure 1). This structure meets the high speed computational requirements for attitude control associated with strapdown systems, and also achieves the reliability required for long time mission success.

The modular computer requirements have been derived through simulations which yielded speed, word length, and memory requirements.

A breadboard model consisting of two columns has been built and is currently in the terminal stage of system checkout. Software is being developed concurrently with hardware. This Modular Computer Breadboard (MCB) will be used for experimenting with different structures in order to enhance the NASA ERC modular computer objective. The body of this paper describes the LSI implementations of the modular computer, with requirements and organization given in the following sections.

### *The NASA modular computer requirements*[7] *

The functional design requirements can be character-

* A summary is included here for easy reference.

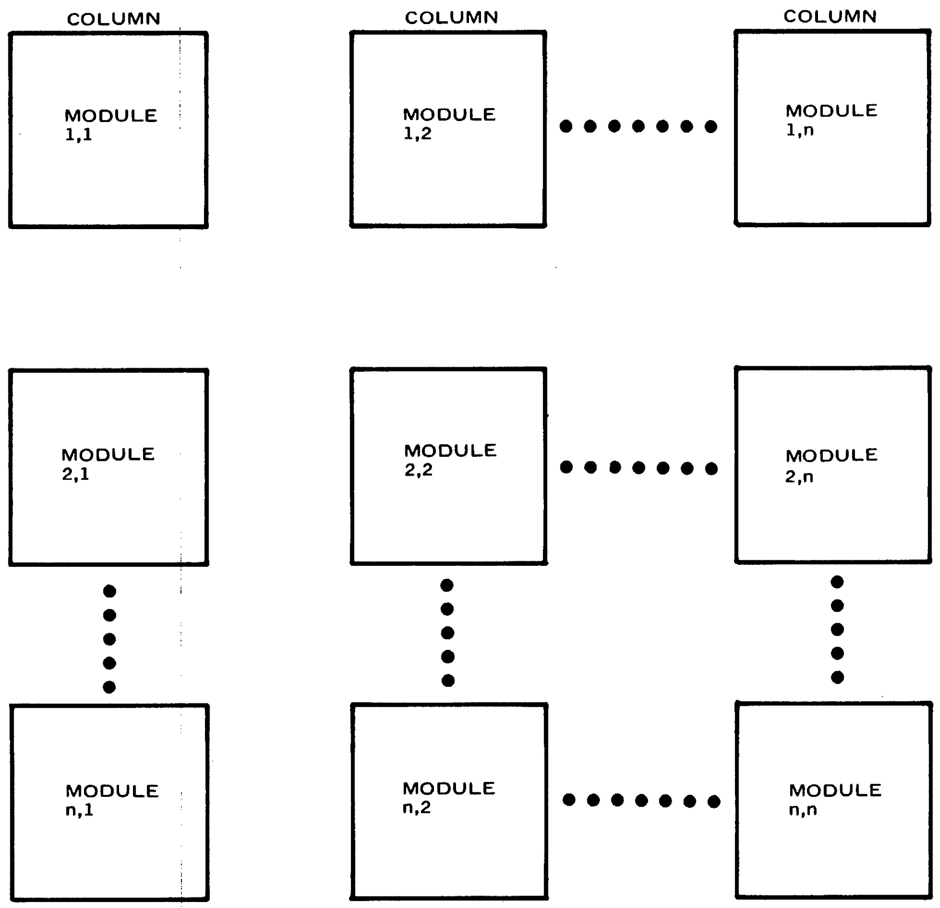


Figure 1—NASA modular computer showing columns and modules

ized by high probability of success over a short period for high speed computations and survival for long periods at low computation rates.

The Modular Computer, as a potential component of a guidance and navigation subsystem of several potential space booster configurations, must be applicable to at least four distinct missions: the synchronous satellite, lunar orbiter, Mars orbiter, and Jupiter fly-by solar probe. Computer memory size, word length, and speed requirements for each phase of these four missions have been estimated by means of computer simulations. The object computer was assumed to have single-address and sequential operation.

Figure 2 shows the computational requirements as a function of injection velocity accuracy. Next to reliability, computational speed is the most critical parameter. Only one set of curves is shown for all missions since it has been assumed that the guidance computational requirements up to and including injection are the same for all missions. The speed (instructions per second) axis represents equivalent additions per second at a rate of 1 multiply equals 6 adds. The memory requirements include approximately 1,400 words for executive and IO operations, for a total of 12,800 words.

In terms of physical parameters, it is estimated that radiation, temperature, and computer operability requirements represent the most cirtical environmental conditions which the modular computer must meet. The proposed trajectories could subject the spacecraft to 3 to 48 hours of 1-MeV electron flux of $10^9$ e/cm$^2$ sec and 80-MeV proton flux of $10^7$ p/cm $^2$sec. Representative calculations of anticipated ambient thermal environments clearly indicate that an environmental control system is needed. The mission time requirement for navigation varies from six hours for the synchronous satellite to 436 days for the Jupiter fly-by. These times pose stringent reliability requirements.

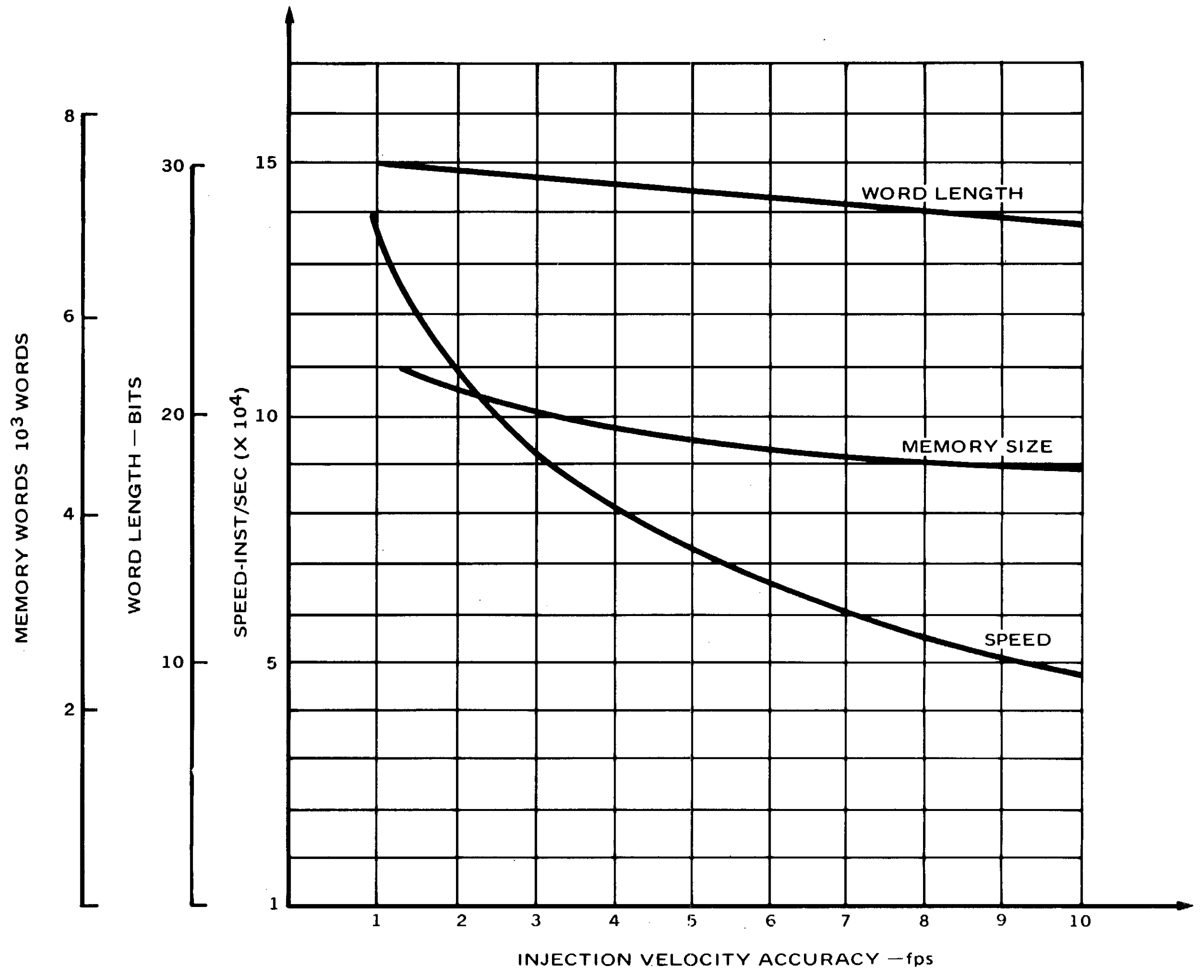


Figure 2—Computational requirements for injection into parking orbit

*The modular computer architecture*

## Design philosophy[8]

The most severe requirements in terms of speed and accuracy occur during boost.[7] Post injection computational requirements are low and the accuracy of computations is far less critical. Therefore, to satisfy the composite requirements a Modular Computer (MC) organization as shown in Figure 3 has been structured. Each column of the MC can satisfy the $1.5 \times 10^6$ instructions/sec requirement.

During boost, three columns of the modular computer operate concurrently in a triple modular redundant (TMR) mode, with majority voting at the outputs. After orbit injection, the TMR mode is terminated and the ensemble of modules is configured so that only one computer remains operating; the others are turned off to conserve power and improve reliability.* System interlocks are provided which insure that the on-computer performs correctly (within bounds). If this is not the case, the Configuration Assignment Unit (CAU) is triggered. It is the task of this unit to assemble at least one computer out of all the available modules.

* The failure rate of non-operating circuits is assumed to be lower than that for operating ones.

The availability of good modules is determined by means of hardware-software tests with interlocks. As may be seen from Figure 3, each of the computers has been separated into four functional modules: a Memory Unit, Control Unit, Arithmetic Unit, and an IO Unit. The Configuration Assignment Unit (CAU) in conjunction with the CU, together with the Configuration Control Switches (CCS), can automatically reconfigure the ensemble so as to form an operating computer. Such a computer may consist of any combination of MU-i, CU-i, AU-i, IO-i.

The breadboard version of the modular computer contains two columns. This is sufficient for the intended experiments:

1. Determination of mission algorithms within

specified accuracy limitations and consistent with the intended application.
2. The use of parallel processing to achieve higher effective computational speed.
3. Automatic detection and isolation of the occurrence of a computer module failure, and automatic reconfiguration to eliminate the effects of the faulty element.

## Computer structure

Although Figure 3 shows a tri-column configuration, the actual flight computer may require additional columns and some configuration adjustment in order to meet the mission time requirements.[9]

In general terms, the modular computer consists of:

k—Configuration Assignment Units (CAU)
one set of Configuration Control Switches (CCS)
m—Control Units (CU)
n—Arithmetic Units (AU)
p—Input Output Units (IOU)
q—Memory Units (MU)
r—Power Supply Systems

The values of k through r are determined from reliability requirements and configuration alternatives. In the preliminary design, $k = r = 1$ and $m = n = p = q = 3$. This configuration will be adjusted as required.

### The configuration assignment unit (CAU)

The Configuration Assignment Unit controls the switches which interconnect the various modules to produce the necessary computer or computers. The CAU monitors CU requests for changes in the computer's configuration and, based on a predefined test, may accept or reject these requests. It determines if no operating "computer" exists, and then establishes new configurations until a working "computer" is assembled. The CAU contains registers which permit communication between control units. CU interrupts are generated in the CAU by means of Status and Mask registers. The system clock is also located in the CAU. The primary tasks of the CAU are:

1. To validate requests for change from a CU by monitoring the elapsed time and the result of a diagnostic, and then accepting and implementing the request.
2. To connect all possible configurations one at a time until one operating computer is found, based on diagnostics.
3. To initiate an interrupt in a newly configured computer to start a diagnostic.
4. To provide and monitor a counted delay of about 30 seconds which, if not reset in time, will be interpreted as the absence of a working computer, which will initiate two above.
5. To maintain configuration and status information during a shut-down if power is maintained to the CAU. When power is restored, the two previously stored configurations will be exercised first to locate an operating computer. If these fail, 2 above is initiated.
6. To accept from the executive CU requests for changes in IO configuration.

### Configuration control switches (CCS)

As seen from Figure 3, the CCS's provide a path between *any* module in a row with *any* module in rows immediately above and below. In addition, the switches provide for traffic between the CU and IOU modules. All paths are under the control of the CAU.

### Control unit (CU) and Arithmetic unit (AU)

The Control Unit determines the sequence of operations within the computer, which consists of one or more MU's, one AU, and any applicable IOU; i.e., all computer memory, arithmetic, and input/output operations are under the control of the CU. As is seen from Figure 3, the traditional ACP (Arithmetic and Control Processor) has been split into separate functions of CU and AU. This is done to enhance processing speed and long term reliability. Each unit has a set of 16 temporary registers much like the multi-usage registers of third generation computers, except that there are three index registers which are separate and distinct in addition to the temporary registers. The AU and CU operate concurrently. The AU accepts data and instructions from the coupled CU and executes these instructions under internal control, making the results available to the same CU. Two's complement arithmetic, both floating and fixed point, are included.

### The input output units (IOU)

The Input Output Units (IOU) are of the direct memory access type, which provide cycle-stealing access for IO transfers. Each IOU provides two input and two output channels. The IOU contains two registers which can be loaded by the CU. These registers hold the priority and normal operation control words. When

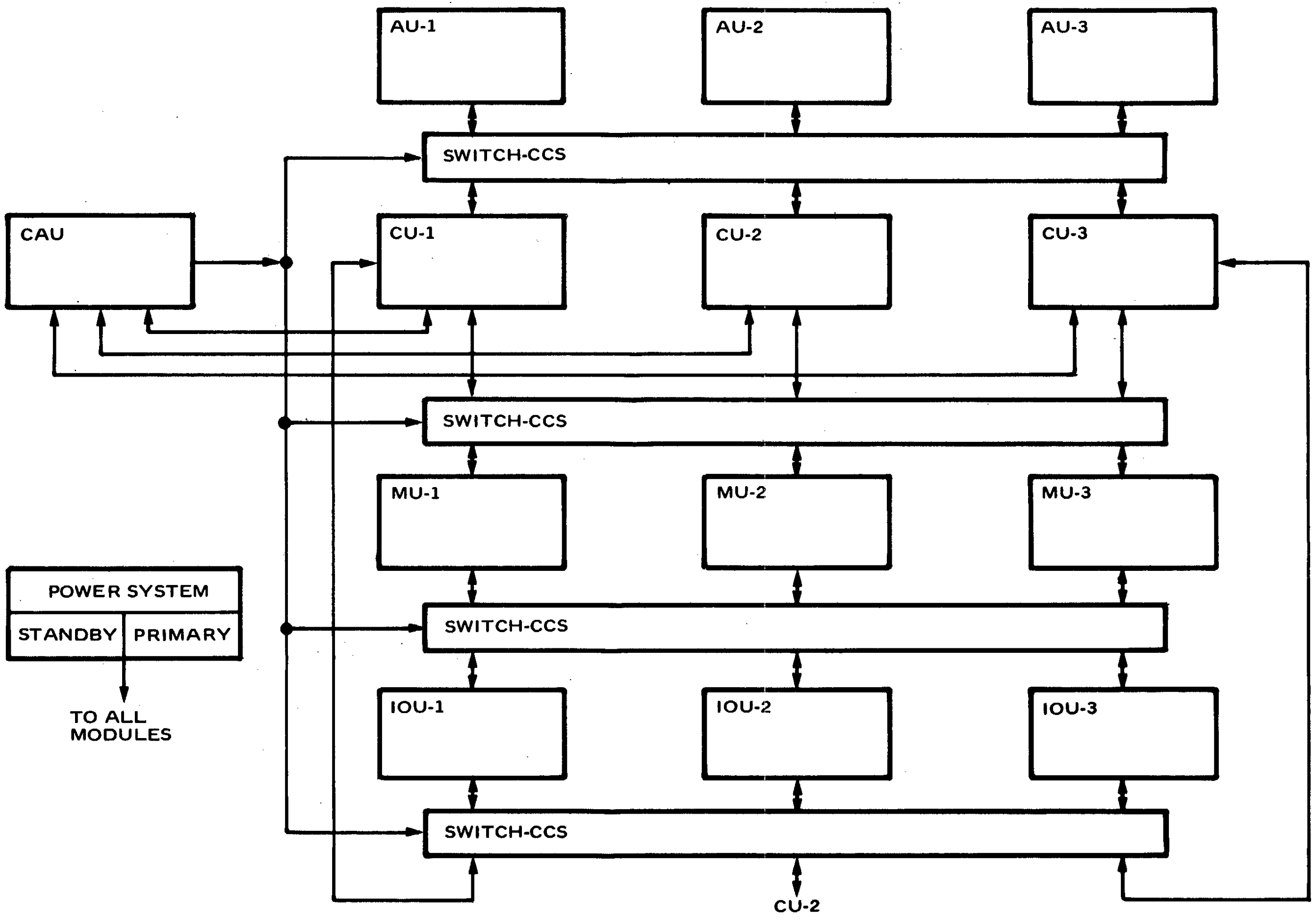


Figure 3—Modular computer organization

data for the priority channel is absent, the normal transaction is served.

The memory unit (MU)

The Memory Unit (MU) receives, parity checks, and stores incoming data in the assigned address. The address is also checked for parity. At present, each memory unit can store 4,096 words of 36 bits each. A read-restore cycle is completed in 1 microsecond. Each memory is addressable by any CU, as permitted by the CAU. Two CU's are not allowed to be associated with one MU. The CAU may permit a CU access to more than one memory. Memory access is through a combination of sequential and priority control. First access is assigned to data from the IOU, while second priority is assigned to the CU.

*LSI implementation of the modular computer*

### Overview

Size, power, and reliability constraints demand that the modular computer be implemented with LSI circuits, but the question of how to achieve an LSI implementation remains. To date, several approaches to logic partitioning for LSI have been reported, ranging from the conventional approach, where partitioning is done after the logical equations have been written, to the "cellular" type approach, where a group of logical gates are structured to be programmed on the cell to form specific functions.[9,10,11,12]

The conventional approach includes both manual and automatic partitioning. This approach appears undesirable for the modular computer implementation because the design process tends so be lengthened[13] and the number of LSI chip types tends to increase, particularly as applications are broadened outside of the computer proper. A small number of LSI chip types is an important factor towards achieving the very tight quality and process controls required for the realization of very low component failure rates. The latter is a must for long time mission reliability.

In the cellular approach, the cell design is such that all combinations of n variables must be implementable in order for the cell to be of universal use. Proofs have been developed showing that such a cell can indeed form all functions of n variables. The cell, although a universal device, still requires the process of writing logic and determining which paths in the cell structure should be connected or cut (physically or logically) in order for the universal cell to assume the unique logic posture specified by the logic designer.

A functionally organized set of building blocks with predetermined* logic interconnects has been chosen for the modular computer implementation. This set, called functional characters, tends to satisfy the requirements of a small number of LSI types.

The set of characters, 10 in all, was selected through a pragmatic approach to logic partitioning. As for the cellular technique, the characters have predetermined logic interconnects but do not require restructuring of interconnections in order to achieve the logical design objective. The design process with functional characters is analogous to programming using a compiler. The characters are analogous to compiler statements. The designer specifies inputs, outputs, and control for each character's micro–operation. Micro-programming is used as the control structure. Three of the 10 characters comprise the micro-program store. Perhaps designing with pre-specified large functions without the utilization of Boolean equations marks the greatest departure and contribution of the functional characters.

No attempt is made to demonstrate that a character or the set can implement all combinations of n variables. All combinations are not required in order to build effective computing machines. The design philosophy permits the introduction of new characters if the existing ones are shown to be ineffective.

The ten functional characters exist as logical blocks containing approximately 350 gates per block. These blocks can be subpartitioned into smaller blocks with fewer gates per block or chip, whereby several smaller blocks would compose a functional character (see Table V). A reduced-width set of functional characters has been breadboarded using conventional IC circuits. This demonstrated the modularity and versatility of the characters.

The characters can be implemented with LSI circuits, using cellular or threshold logic, or any other appropriate technique. An overview of the characters is presented here.** Statistics are given comparing the functional character design of the Modular Computer Breadboard with the implementation utilizing custom logic design and partitioning, as found in the implemented Modular Computer Breadboard (MCB). Regrettably, there is no means for a one-to-one comparison using identical stages of MCB implementation. To the extent practical, the comparisons address the same system parameters. The comparisons assume that all cards of MCB containing IC's have been converted to equivalent LSI chips.

## Description of the functional characters

The functional character set is a group of logic arrays forming a self-sufficient family of building blocks that reduce computer design to a determination of character types and number, followed by micro-programming of the set. Ten character types have been shown to be sufficient for the building of both special purpose and stored program general purpose digital equipments. These characters are:

| | | |
|---|---|---|
| G1 | Register storage | |
| P1 | Scratch pad memory | |
| L1 | General logic | |
| L2 | Arithmetic logic | |
| L3 | Input/Output | |
| M1 | Micromemory sequencer | Microprogram memory |
| M2 | Micro-instruction Register | |
| MM | Micromemory array | |
| P2 | Up/Down counter | |
| P3 | Switch | |

Tab'e I shows the gates, pins, and gates/pin ratios for these functions.

Characters of the same letter are logically grouped into a common unit, as illustrated in Figure 4. This arrangement extends the register count and word length. The complexity of logical operation can also be extended by the cascading of characters. Several microprogram strings can be executed simultaneously. The micromemory function was divided into three

* The logic of the block is designed prior to the computer design

** More detailed discussion on the subject is found in paper by F. D. Erwin and J. F. McKevitt. of this Proceedings.

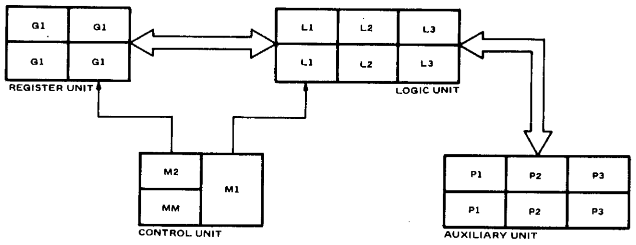


Figure 4—Typical functional character configuration

characters in order to provide for greater versatility. The array can be adapted to different size programs. The instruction register may be cascaded using two or more M2 characters, and still operate under a single sequencer control.

TABLE I—Composition of the ten character types sufficient for building special purpose and stored program GP digital equipments

| | Gates | Pins | G/P |
|---|---|---|---|
| G1 General Register | 224 | 62 | 3.4 |
| P1 Scratch Pad | Depends on system architecture (8 × 16) bits/block | | |
| L1 Boolean | 274 | 145 | 1.8 |
| L2 Arithmetic | 250 | 77 | 3.3 |
| L3 I/O | 377 | 149 | 2.5 |
| M1 Sequencer | 348 | 91 | 3.8 |
| M2 Instruction | 323 | 131 | 2.5 |
| MM Array | Depends on size of program 2048 bits/block | | |
| P2 Up/Down Counter | 147 | 81 | 1.8 |
| P3 Switch | 210 | 118 | 1.8 |

## Functional character implementation of the modular computer breadboard (MCB)

The functional character appears to have a broad range of applications. This was demonstrated in the study by implementing an A to D, DDA, and the modular computer.[15] For the purpose of evaluation, the breadboard version (MCB) was implemented using the functional characters. The MCB is a two column configuration of the modular computer. The functional and operational aspects of the MCB have been preserved in the functional character implementation. However, the implementation detail was tailored to the functional character set. This includes the grouping of registers into memory arrays and complete microprogramming, which are not part of the MCB.

Figure 5 shows the block diagram of the existing MCB. This diagram has been overlaid with the character implementation as shown in Figure 6. Note the P3 blocks of Figure 6 are equivalent to the switches (CCS) of Figure 5. In the block diagram form, the MCB implementation using functional characters is depicted as an assemblage of characters each under microprogram control. The microprogram resides in the micromemory, which consists of the MM, M1, and M2 characters. The word length is determined by the number of juxtapositioned characters of the same type. In general, the characters are 8 bits wide. The P1 character is 16 bits wide. The number of G1 or P1 rows identifies the number of registers of the P or G type. The G type operates directly under microprogram control, whereas the P type operates indirectly under microprogram control. The G1 character contains four registers for a 4 × 8–bit array. The P1 character contains 16 registers for a 16 × 16–bit array.

Figure 6 also shows the character content of each module adjacent to name of the module. The number to the left of the slash (/) is the total number of characters used per module, regardless of type. These numbers represent the first microprogram pass referred to in Table III. The number to the right of the slash is the number of character types used in each module regardless of the number of modules. Note that the number of characters is additive, whereas the number of character types is not; the sum of the character types is 10.

## Evaluation of the functional character design

Table II shows the comparison data of the functional character implementation versus the existing MCB implementation. As may be seen, in all aspects, except gates committed**, the functional character implementation results in a significant improvement over the existing MCB design. The number of gates committed is 35 percent higher for the functional character approach. In the LST area, the tradeoff will no doubt recognize the functional character approach as significantly superior. An increase of 35 percent in the number of gates committed is a small price to pay for the reduction in the number of chip types and pins.

As will be shown later for reliability purposes, a small number of pins in the system is far more im-

** "Committed" rather than "used" is the proper descriptor since some gates on the chip or conventional card are unused but yet they are committed by virtue of being part of the chip or card.

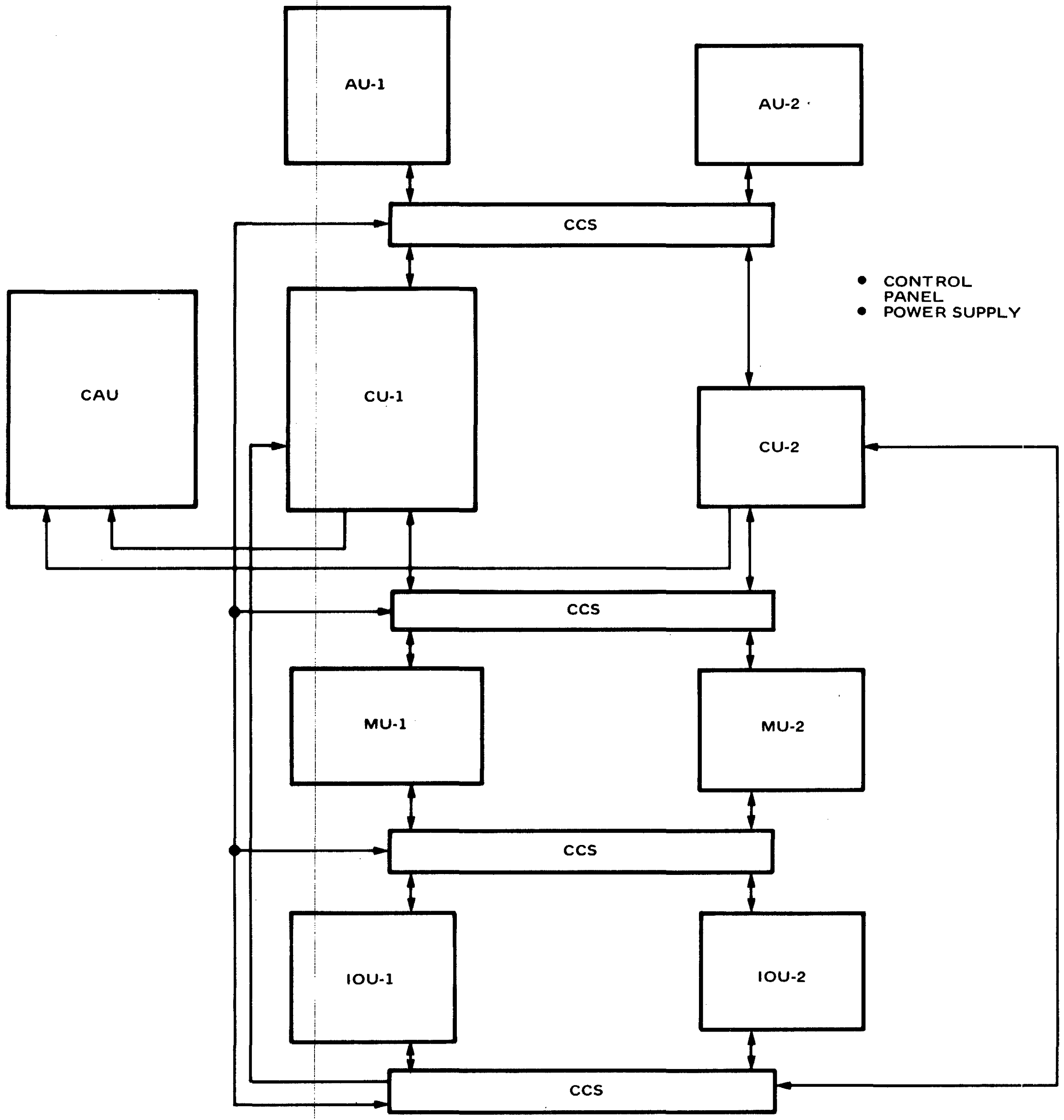


Figure 5—Block diagram of existing modular computer breadboard overlayed with the character implementation shown in Figure 6.

portant than a small number of gates, all other factors being equal. As seen in Table II, the number of pins required for the MCB implementation is 2.6 times the number required for the functional character implementation.

As the column heading shows, the comparison in Table II is made between two LSI implementations: one representing the functional character technique the other representing the conventional approach where every MCB card containing X number of IC's has been converted to an equivalent IC with the number of card terminals becoming the equivalent LSI package pins.

The implementation with the functional characters

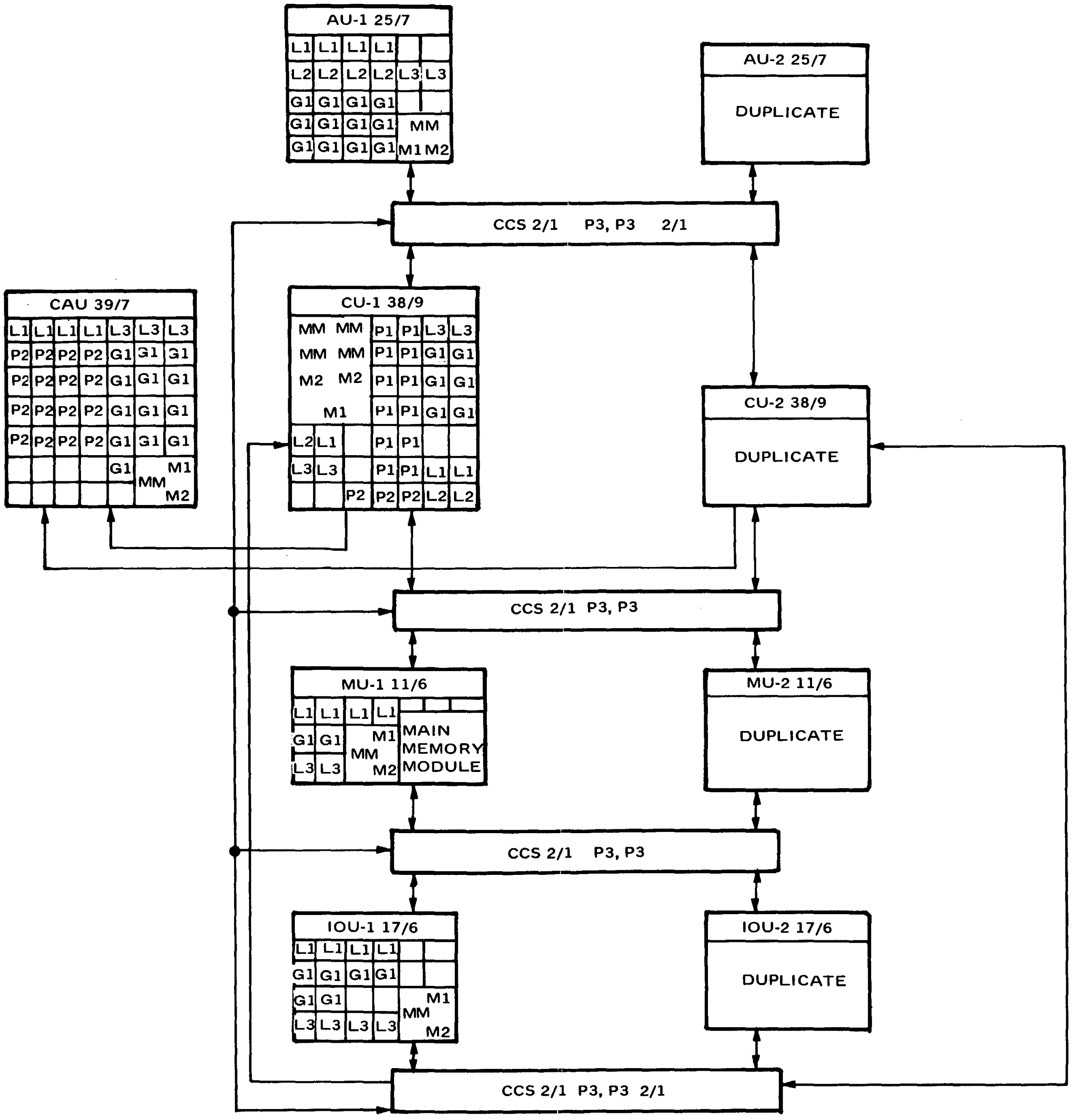


Figure 6—Functional character implementation of MCB

resulted in a 35 percent greater throughput. This is because the functional character assumed a 32 percent faster gate. For equal gate delays the two implementations would yield approximately equal throughput.

The most significant point from a quality control point of view is that the entire computer was implemented with ten character types—three of these belong to the "micromemory" domain used for microprogramming of the computer modules. The micromemory array (MM) is the storage element which contains the control information. If permanent memory is used, it may be necessary to generate the desired information content on a number of different chips. However, effort is being expended in industry towards producing electronically alterable, read only memory arrays.[17] Progress to date shows that there is promise of being

TABLE II—Comparison of functional character implementation and existing MCB implementation[16]

| Item | Implementation Functional Character Implementation (Units) | MCB Assuming Each Card Is An LSI Chip | Percent Improvement Over MCB Implementation | High/Low Ratio |
|---|---|---|---|---|
| Types | 10 | 23 | +56 | 2.30 |
| Cards (LSI Chips) | 206 | 554 | +63 | 2.70 |
| Pins Committed | 18,200 | 47,600 | +62 | 2.62 |
| Gates Committed | 47,200 | 35,000 | −35 | 1.35 |
| Gates/Pin | 2.6 | 0.75 | +250 | 3.47 |

TABLE III—Effects of microprogram improvement on the functional character implementation of the MCB

| | No. of Characters Used | | No. of Character Types Used | |
|---|---|---|---|---|
| | MICRO-PROGRAM PASS | | MICRO-PROGRAM PASS | |
| Unit | First | Subsequent | First | Subsequent |
| MU | 11 | 7 | 6 | 5 |
| CAU | 39 | 38 | 7 | 7 |
| CU | 38 | 35 | 9 | 9 |
| AU | 25 | 21 | 7 | 7 |
| Switches | 8 | 8 | 1 | 1 |
| I/O | 17 | 17 | 6 | 6 |
| Computer Total System | 229 | 206 | 10 | 10 |

able to use only one array with identical metalization patterns. This array will be encoded with the proper information content at the time of use.

It is reasonable to project that ten characters and ten masks are sufficient to implement the MCB and the majority of digital equipments. Other types of equipment were implemented, including A to D and D to A conversion logic and a DDA. All designs utilized the same characters but different microprograms. The efficiency of gate usage was best in the MCB implementation and worst in the DDA.[15] It is premature to conclude that a different character is required for a more efficient implementation of the DAA. The MCB design was optimized through remicroprogramming, but this was not done with the DDA and A to D equipments.

Design with functional characters saves time. During a six month period, the entire MCB was designed, microprogrammed, and remicroprogrammed several times. This illustrates the ease and speed of the design process. The improvements gained through microprogramming are demonstrated in Tables III and IV.

Table III shows the improvements in terms of the number of characters and character types required for the two microprogram passes. The characters remained unchanged. In this comparison, the configuration of the MCB was identical with the presently implemented IC version.

Further improvements were gained, as shown in Table IV, by restructuring the MCB with the appropriate remicroprogramming. The AU and CU were combined into one unit, eliminating some logic and the switch between them. This reimplementation was feasible with the functional character set due to the more general nature of the characters as contrasted with the custom implementation of the existing MCB.

TABLE IV—Effects of combining the AU and CU of MCB

| Parameter | Functional Character Implementation of the Existing Configuration | Same Except AU and CU Were Combined |
|---|---|---|
| No. of Characters | 229 | 182 |
| No. of Character Types | 10 | 10 |
| Fixed Point Direct Add | 9.9 us | 4.2 us |
| Fixed Point Add | 11.6 us | 6.4 us |
| Fixed Point Subtract | 11.6 us | 6.4 us |
| Inclusive or | 11.5 us | 6.2 us |
| Exclusive or | 11.5 us | 6.4 us |
| Logical and | 11.5 us | 6.2 us |

Combining the AU and CU into one unit may affect the long term reliability. This and curiosity about the relative merits of multiprocessor structures, such as the Hughes H4400 (currently being built), vs. modular computers, such as the MCB, led Hughes to study factors affecting long term reliability. In this study, modules of equal complexity, with the exception of the switches, were assumed. The results are presented in Reference 9.

Several interesting points are worth mentioning here:

1. Multiprocessors have an improved short term reliability, but the long term reliability is degraded somewhat.
b. Different configurations, or organizations, significantly affect long term reliability.
c. Component reliabilities (failure rate of the characters) markedly affect the mission reliability.
d. The failure rates quoted for existing IC's of $10^{-8}$ failures per gate-hour will have to be significantly reduced in order for either the multiprocessing or the modular computer organization to reach the desired long time mission reliability objectives.

*Circuit realization of the functional characters*

This section presents circuit considerations for the LSI realization of the functional characters. The circuits must not only reflect the correct logical functions but also, because of the potential space applications, satisfy the electrical, thermal, and mechanical constraints.

The circuit solutions are to be designed to reflect a set of NASA design guidelines that are intended to insure a high probability of mission success. These guidelines are:

| | |
|---|---|
| Gates per chip | —About 100, no more than 150 |
| Circuit yield | —100% without yield enhancement |
| Conductor spacing | —0.1 mil minimum |
| Conductor width | —Current density not to exceed $10^5$ amps/cm$^2$ |
| Metalization layers | —No more than 2 |
| Circuit type | —Bipolar TTL |

The 100 gate per chip function size limit reflects the 100 percent yield and TTL technology constraints. It is expected that LSI and TTL circuits containing about 100 gates will be producible with 100 percent yield. Other circuit technologies such as MOS may accommodate a larger number of gates per chip.

As may be recalled from Table I, some functional characters require about 350 gates per function. The natural tendency would be to implement one character per chip. However, this is not an acceptable solution for TTL circuits in view of the above constraints. Therefore, the functional characters were subpartitioned as shown in Table V.

The intent of the table is not to select the optimal subpartition, but to enumerate some logical choices. The optimal choice will depend on assigned weightings for gates and pins per chip, as well as the other design constraints mentioned earlier. The table thus shows each character and the characters' composition, using one or more custom or commercially available LSI/MSI chips. More than one subpartitioned chip is required to implement the functional character. The number of chips and chip types required is given in the second column as a descriptor and also in the sixth and seventh columns under "composite." The columns under the "composite" heading state the total number of items required to implement one functional character. The columns under the first and second chip heading contain similar information on a per chip basis.

A comparison of Tables I and V shows the following changes:

1. The number of chip types is at least 20% greater than the number of characters; thus, paying a small penalty in terms of part number problems.
2. The number of gates per chip dropped (approximately by a factor of 0.5) and the number of pins remained about equal, resulting in an increased number of pins in the system by a factor of about 2.
3. The total number of gates per function increased an insignificant amount.

As is shown below, these changes tend in the wrong direction for obtaining improved MTBF's of the modular computer. As is seen from the above and Table V, the subpartitioned characters would require a greater number of bonds (pins) and will therefore operate at higher temperatures than the non-subpartitioned set. The temperature rise is due to the increased number of gates required and the higher current required due

TABLE V—Alternate schemes for sub-partitioning

| Character | | Composite | | | | | 1st Chip | | | | 2nd Chip | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Name | Composition | Gates | Pins | G/P Ratio | Chips/ Character | Chip Types | Gates/ Chip | Pins/ Chip | Ratio Gate/Pin | No. Used | Gates | Pins | Gate/Pin Ratio | No. Used |
| G1-Register | 2 custom chips, single type | 224 | 62 | 3.6 | 2 | 1 | 112 | 52 | 2.2 | 2 | | | | |
| L1-Logic | 2 custom chips, single type | 274 | 145 | 1.9 | 2 | 1 | 137 | 138 | 1.0 | 2 | | | | |
| L2-Adder | 2 Identical custom chips | 258 | 77 | 3.4 | 2 | 1 | 129 | 60 | 2.2 | 2 | | | | |
| | 1 custom and 1 commercial chip | 228 | 77 | 3.0 | 2 | 2 | 117 | 88 | 1.3 | 1 | 111* | 43* | 2.6* | 1* |
| L3-Input/Output | 4 identical chips | 456 | 150 | 3.1 | 4 | 1 | 114 | 72 | 1.6 | 4 | | | | |
| | 2 identical chips with optional parity | 410 | 149 | 2.8 | 3 | 2 | 150 | 87 | 1.7 | 2 | 110 | 68 | 1.8 | 1 |
| | Alternate Scheme 2 chips + optional parity chip | 398 | 149 | 2.8 | 3 | 2 | 150 | 95 | 1.6 | 2 | 98 | 66 | 1.5 | 1 |
| | Optimal 3-chip configuration | 377 | 149 | 2.5 | 3 | 2 | 129 | 85 | 1.5 | 2 | 119 | 86 | 1.4 | 1 |
| M1-Micromemory Sequencer | 3 chips-2 types | 358 | 91 | 3.9 | 3 | 2 | 150 | 92 | 1.6 | 1 | 104 | 85 | 1.2 | 2 |
| | 3 chips-2 types 150 gates if I.C. | 348 | 91 | 3.8 | 2 | 2 | 142 | 73 | 1.9 | 1 | 206 | 73 | 2.8 | 1 |
| P2-Counter | 1 custom chip | 148 | 81 | 1.8 | 1 | 1 | 147 | 81 | 1.8 | 1 | | | | |
| | 1 custom and 2 commercial chips | 163 | 81 | 2.0 | 3 | 2 | 40* | * | * | 2* | 83 | 82 | 0.9 | 1 |
| P3-Switch | 2 identical chips | 210 | 118 | 1.8 | 2 | 1 | 105 | 75 | 1.4 | 2 | | | | |
| M2-Micro Instruction Register | 3 chips-2 types | 323 | 131 | 2.5 | 3 | 2 | 100 | 51 | 2.0 | 2 | 123 | 89 | 1.4 | 1 |

*Commercially available chip

to a larger number of external gates.* More pins require more external gates to drive the capacitance of the external pins. Both factors, increased pins and higher temperature, increase the failure rate of the device and thus lower the probability of mission success.

Reliability considerations require a minimum number of bonds (pins) and a lowest junction temperature practicable.[18] Several other factors affect reliability. These are either less influential on the operational failure rate, or on a relative basis do not affect the tradeoff. For example, the quality of the package's hermetic seal may be an important factor in development and acceptance testing. But once a good seal has been established, it will remain good. Furthermore, the difficulty of making a good seal is proportionate to the lengths of seal interface. The latter in turn is a function of the number of pins per package, which for the cases in question is about the same.

* LSI circuits are generally built with tailored lower power in ternal gates for driving low capacitance and limited fanout within the chip's boundaries and higher power gates at the chip output in order to overcome the input output capacitance and chip fan-out.

Temperature is a very important consideration since the failure rate of the device increases about 1.8 times per 25°C temperature rise.[18]

Within specific cooling capacity, circuits, and packaging technology, two factors affect the device's temperature:

a. The number of gates per system.
b. The number of IC package pins per system.

For example, in the natural and subpartitioned functional characters (Tables I and V) the number of gates per system remains approximately constant. However, the number of pins nearly doubled for the subpartitioned case. Typically, in TTL circuits the power dissipation of the subpartitioned implementation is expected to increase. Specifically, the dissipation is increased by a factor of 1.08. Using the data from Table II, the total number of functional character gates in the MCB is 47,200 and the number of pins is 18,200.

Assuming a power dissipation p and 2p or more for internal and external gates, respectively, the power dissipation for the MCB is:

$$P = p\left(M + \frac{N}{3}\right) = 53{,}300\,p^{**}$$

where

$M$ = total number of gates

and

$N$ = total number of pins

P for the subpartitioned implementation is 57,700 = p (47,200 + 10,500).

Using the same formula, the power dissipation for the functional character implementation (Table II) versus conventional MCB implementation is 53,300 p versus 50,900 p, respectively.

Even though the number of gates is 35 percent greater for the functional character implementation, the power dissipation is about 5 percent greater than that of the MCB's, were it implemented with LSI's representing present MCB cards. This 5 percent difference will disappear in practice. The octual power difference relative to the present IC implementation would be in favor of the functional implementation.

In addition to the number of pins causing increased power dissipation, which may be equated with increased failure rates, there are other reliability and cost penalties associated with an increased number of pins. These all result from bonding. Each pin requires two internal bonds (one to the metalization, the other to the pin). Each pin must in turn be fastened to some external holder (card, connector, wire, etc.).

Every one of these junctions is a potential failure and a fabrication cost factor. Thus, the number of pins as a contributor to increased system failure rate manifests itself in several ways. Every effort must be made to keep the pin count low.

The "ideal" LSI chip, assuming it could be built would contain the largest number of gates and use the lowest speed power product circuit. Figure 7 shows the various circuits currently available and the speed-power-product lines (PL)[19]. Note that the "ideal" circuit for space applications would be located in the lower left corner of the figure. The ion implanted and complementary MOS circuits come closest to the "ideal" circuit. The shaded area shows the speed-power coverage of the P channel ion implanted MOS (IMOS). The area for the N channel IMOS is forecast to be below that shown for the P-IMOS. At the speed considered the complementary MOS would straddle the P and N areas. The complementary circuit is attractive as a compromise speed–power option. However, it requires about twice as many devices per circuit over single channel. Thus, a single chip would be unable to support a complete function, resulting in increased pins per system. This is undesirable, as pointed out earlier.*

From this, we conclude that the ion implanted MOS type circuit (single channel, high speed, low power) is optimal for the functional character implementation of the MCB, barring producibility problems. It provides the desired density at 100 percent yield, lower power dissipation, and desired circuit producibility.[19,20,21] There are not sufficient practical data to make a judgment. If the "ideal" circuit is not available, a meaningful system can be built using TTL circuitry for the functional character implementation. The penalty is increased power and pins required at a very significant gain of availability of proven circuit technology.

## CONCLUSIONS

It has been demonstrated that digital equipments can be designed using pre-specified logical building blocks called functional characters. Once the logical design of the functional character has been accomplished, the system designer no longer needs to employ Boolean equations to specify the system. He needs only to specify the inputs and outputs of the characters and microprogram the sequence of their operations. The set of functional characters can be considered as standard and "universal" LSI chips that are sufficient to implement most digital equipments. Two desirable features of the characters are that the number of chip types and pins in the system are significantly reduced.

It may be inferred that standard design automation programs which have as inputs Boolean statements or their equivalent will not be applicable as functional character design aids. Routing programs have the greatest potential of being useful. Simulation programs will have to operate at a macro level. A microprogram assembler is a desirable program.

In order to obtain the required $10^5$ hours between system failures, it will be necessary to improve the system configuration of the modular computer and to improve the basic circuit or module reliability. The

** Each pin must require at least one external gate, and each external gate dissipates at least p more units of power. Typically, ⅓ of the pins are used for output; the others are used for inputs, power, and ground.

* The ratio of power dissipation for internal and external gates is much greater for MOS at the desired speed.

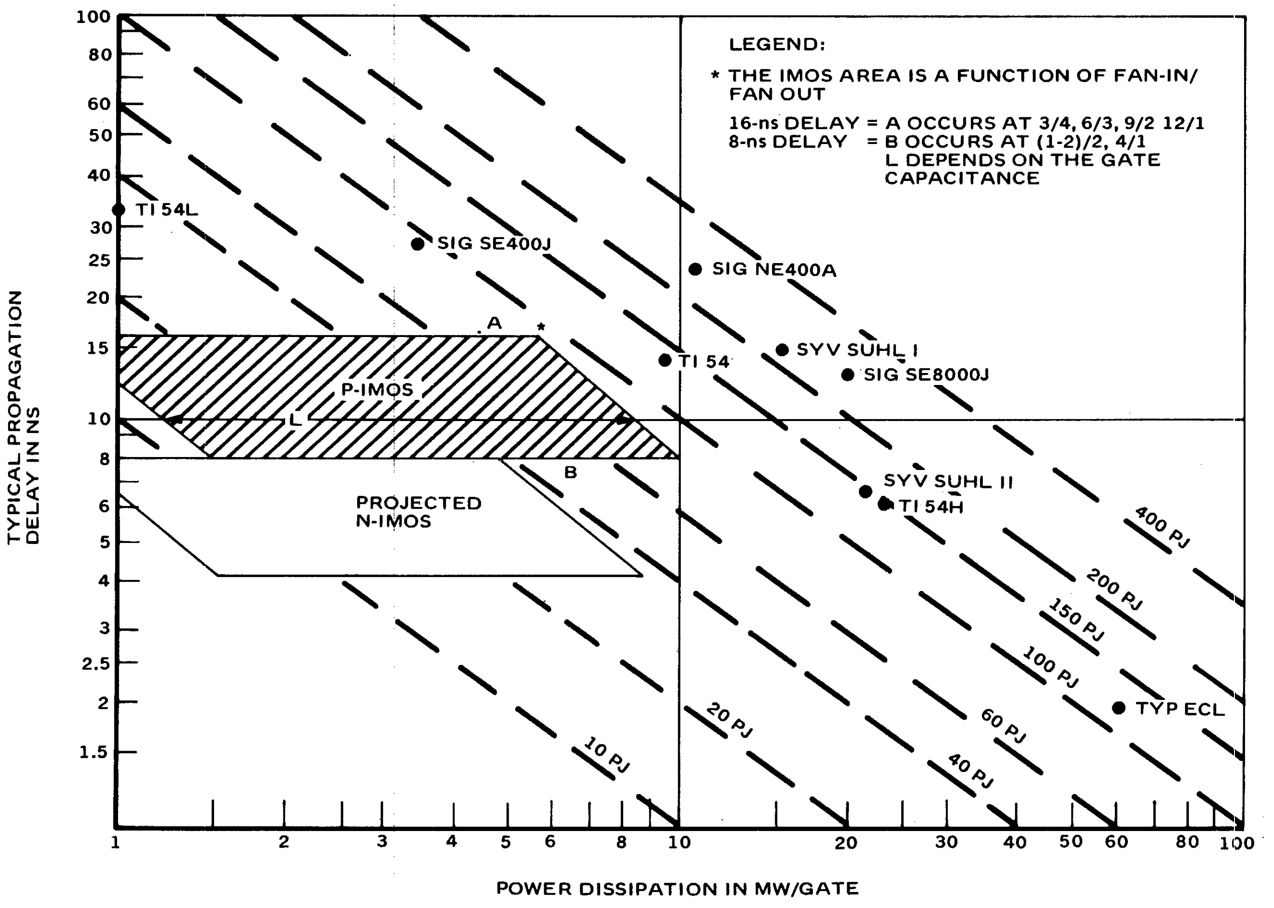


Figure 7—Speed-power products of some bioplar and MOS circuits

functional character implementation of the modular computer will readily allow configuration changes. The module content and overall system configuration can be readily changed. The characters improve the module's MTBF because of the significant reduction in the number of pins.

The reduced number of chip types facilitates quality control, thereby potentially improving the module's MTBF.

Any circuit or chip wiring technique can be used to implement the characters. Currently, the modular computer is planned to be implemented with TTL, 100 percent yield LSI technology. Other circuits and technologies are being evaluated.

## ACKNOWLEDGMENT

The research reported in this paper was sponsored in part by the Electronics Research Center under Contract NAS 12-665.

The authors express their appreciation to Mr. W. L. Martin of Hughes Aircraft Company for his many suggestions for improving this report.

## REFERENCES

1 D O BAECHLER
*Trends in aerospace digital computer design*
Computer Group News Vol 2 No 7 Jan 1966 18–32

2 *A study of Jupiter fly-by-missions*
General Dynamics Rpt FZM-4625 May 17 1966 3–159 to 3–202

3 A AVIZIENIS
*Design of fault-tolerant computers*
Proc FJCC Vol 31 1937

4 M M DICKINSON J B JACKSON G C RANDA
*Saturn V launch vehicle digital computer and data adapter*
Proc FJCC Vol 26 1964 501-516

5 G H BARNES R M BROWN M KATO
D J KUCK D L SLOTNICK R A STOKES
*The ILLIAC IV computer*
IEEE Trans on C Vol 17 No 8 Aug 1968

6 E J DIETERICH L C KAYE
*A compatible airborne multiprocessor*
In this Proc

7 H E MAURER R C RICCI
*Horizons in guidance computer component technology*
IEEE Trans on C Vol 17 No 7 July 1968
8 E H BERSOFF E HOPE F TUNG
*IEEE transactions on aerospace and electronic systems*
To be published
9 F D ERWIN E H BERSOFF
*Modular computer architecture strategies for long term missions*
In this Proc
10 R C JENNINGS
*Design and fabrication of a general purpose airborne computer using LSI arrays*
IEEE Computer Group Conf Digest June 1968
11 H R BEELITZ S Y LEVY R J LINHARDT H S MILLER
*System architecture for large-scale integration*
Proc FJCC Vol 31 1967
12 R C MINNICK
*Cutpoint cellular logic*
IEEE Trans on EC Dec 1964
13 R C MINNICK
*A survey of microcellular research*
Journal of the Association for Computing Machinery Vol 14 No 2 April 1967
14 J J PARISER
*Connection considerations with a view toward batch fabrication*
Proc Nat Symposium of the Impact of Batch Fabrication on Future Computers April 1965
15 J J PARISER F D ERWIN J F McKEVITT J A BURKE C P DISPARTE
*Research in the effective implementation of guidance computers with large scale arrays*
First Interim Rpt Submitted to NASA ERC Oct 1968
16 J J PARISER F D ERWIN J F McKEVITT C P DISPARTE J A BURKE
*Research in the effective implementation of guidance computers with large scale arrays*
Second Interim Rpt Submitted to NASA ERC 1969
17 H G DILL R W BOWER K G AUBCHON T N TOOMBS
*Anomalous behavior in stacked-gate MGS tetrodes*
International Solid State Circuit Confetence, Phila 19-21 February 1969
18 G R VAN HOODE
*Evaluation of experience with micro-electronic integrated circuits*
TRW No 9990-6183-R000 May 1967
19 J SEGAL
*Speed/power chart for digital IC's*
The Electronic Engineer June 1968
20 R W BOWER H G DILL K G AUBUCHON S A TOMPS
*Characterization of MOS FETs formed by gate masked ion implantation*
Given at the Internat Electron Devices Meeting Wash Oct 1967
21 H G DILL
*Offset gate field effect transistors with high drain breakown potential and low miller feedback capacitance*
IEEE Trans on Electron Devices Oct 1968

# Project DARE: Differential Analyzer REplacement by on-line digital simulation

*by* GRANINO A. KORN

*University of Arizona*
Tuscon, Arizona

## INTRODUCTION

While batch-processed applications of convenient, highly developed digital continuous-system simulation languages are now commonplace,[1,2] such systems do not provide the intimate man-machine intercourse cherished in analog/hybrid simulation. The DES-I system,[2] which combined a special simulation console and a digital plotter with an SDS 9300 (medium-sized) computer was, then, a pioneering effort, unfortunately abandoned by its manufacturer. The only commercially available interactive system appears to be the IBM CSMP 1130 system which, like its predecessor PACTOLUS,[2] can be programmed from a simple typewriter terminal. This is an interpreter system implemented on a small computer and thus yields relatively quite slow execution.

The writer has felt quite strongly for some time[6] that digital on-line simulation is ready to go—we do have simple simulation-language programming, plus very reasonably priced, fast digital computers, plus new graphic displays. All that would seem to be needed was a system design which would combine these items (Table I), with a good deal of human-factors engineering to make the operator happy as well as efficient. Project DARE (Differential Analyzer REplacement), sponsored by the National Science Foundation at the University of Arizona, is a continuing attempt to develop a series of such systems.

Project DARE demonstrates all-digital on-line simulation of dynamical systems. Each DARE system adds a very convenient but still relatively inexpensive simulation console to a small or large digital computer and can replace conventional analog computers in many applications. System equations or block-statements and input data are entered and conveniently edited on a cathode-ray-tube typewriter. Solutions or phase-plane plots appear on a second cathode-ray-tube display; system parameters and initial conditions are readily changed for successive runs; displayed data can be stored for comparisons; programs and results may be printed and plotted for hard-copy report preparation; and automatic iterative operation is possible. With a reasonably fast digital computer, man-machine interaction at the console is rather more comfortable than with even a modern analog/hybrid computer.

DARE I is a flexible CSSL-type floating-point system permitting relatively slow computation with the PDP-9 computer. DARE II is a block-diagram-based system which trades fixed-point operation for relatively very high speed on the small PDP-9, permitting, for instance, real-time flight simulation. DARE III and DARE IV are only in the planning stage and will implement economical and fast floating-point simulation on a time-shared CDC 6400.

A critical study of future possibilities indicates that DARE-type systems could permit flight simulations including 40 Hz frequencies by 1975, but that modern analog computers are still a hundred times faster. Actual present-day practical applications, however, employ really fast (and therefore relatively inaccurate) analog computation so rarely that much analog simulation could well give way to the more accurate, convenient, and often more economical digital methods demonstrated by Project DARE.

## *DARE I: An on-line CSSL-type system*

DARE I software, written for the PDP-9 by J. Goltz as a Ph.D. dissertation,[5] produces a complete floating-point simulation system, including the basic monitor, editor, and loader used also by DARE II. DARE I source language is essentially similar to the SCI-sponsored CSSL.[1] Though basically equation-oriented, DARE I will also implement user-created analog or hybrid blocks as FORTRAN functions.

TABLE I—*A list of requirements for an on-line digital simulation system*

A useful on-line continuous system simulation system must provide for:

1. *Entry of system differential equations* (in equation and/or block statement form).
2. *Entry of data* (system parameters, initial conditions, function tables, etc.).
3. *Entry of simulation parameters* (frame time, communication interval or display sampling interval, maximum computation time, integration routine used, maximum tolerable error in variable-increment integration routines, choice of variables for display).
4. *Editing*, modification, and correction of the above entries.
5. *Display* of state variables vs. the independent variable (usually the time) and against each other (phase-plane plots).
6. *Preparation of hard copy for reports* in the form of printed tables, xy recorder plots, or strip-chart records.

In addition, a sophisticated simulation system must permit "simulation studies," viz.:

7. *Computations based on results from multiple differential-equation-solving runs* (statistics, cross-plots).
8. *Iterative computation*, i.e., repeated runs with system parameters and/or initial conditions recomputed on the basis of preceding runs for optimization, boundary-value problems).

DARE I employs the FORTRAN compiler supplied with the digital computer and will be described in detail in a separate paper.[5]

DARE I accepts *system differential equations* in first-order (state-equation) form. These equations are simply *typed* in FORTRAN notation on the screen of a CRT typewriter at the right of the DARE console (Figure 1). An interactive CRT typewriter pro-

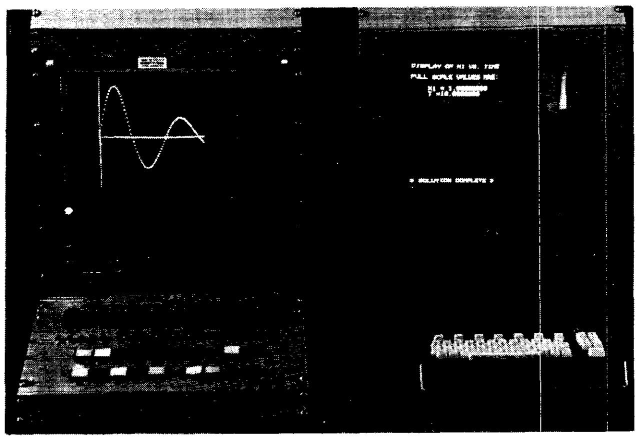


Figure 1—DARE simulation console for use with a PDP-9 or PDP-15 computer. Programs and data are entered, edited, and modified on the CRT typewriter at right. Up to four solution curves, or a phase-plane plot, are produced on-line on the output graphic CRT display at left. A simulation control panel underneath the output display controls simulation and display, with special push-buttons producing hard copy of programs, data, and solutions when desired. The teletypewriter and plotter used for this purpose are not shown.

Console switches (lower left) are sampled by the computer to provide control inputs:

*Method Switch*: A rotary switch used to select the integration routine.
*DT, TMAX, EMAX*: 4-decade thumbwheel switches in an adapted FORTRAN format.
The third decade reads from −5 through 0 to +5, and with the fourth decade indicates a power of 10.
*Elapsed Time*: A strip of 12 lamps to indicate the progress of computation, and to reassure the user that the computer is actually operating when computation exceeds a few seconds.
*Sense Switches*: 2 position switches for various functions, determined by program.
*Trace Finder*: Pushbuttons to identify one of 5 traces on scope display-probably by momentarily blanking it out.
*Command Push-buttons* (lower two rows):
Lighted pushbuttons, for purposes marked on buttons. "Type eqns," "type data," and "select display," are indicators only, *offering suggestions to the user from the computer.* Such suggestions can also appear on the alphanumeric CRT display.

gram proceeds to ask for *problem data* and *simulation parameters*. Of the latter, the frame time DT, the maximum computing time TMAX, and also the error EMAX for variable-increment integration, can be entered either with the CRT keyboard or by console digiswitches, whichever the operator prefers. Console buttons can *recall* selected program or data pages to the CRT screen for editing, or cause them to be *printed out* for report preparation.

As the differential-equation solution proceeds, all state-variable values are read onto DECtape once per "communication interval"[1] (typically every 10 to 50 DT). Thus any selected state variable can be brought back for single or multiple displays and printout; it is possible to compare a current solution with a selected earlier solution display. Permanent graphic records are obtained with an xy recorder and a four-channel stripchart recorder connected to the display.

The choice of *integration routines* for differential-equation solution has been discussed and rediscussed in many survey papers.[2,4] All DARE systems (like the better batch-processing systems[2]) offer a choice of integration formulas. With the on-line systems, console selection of integration routine and frame time (time increment DT) permits very convenient comparison of different integration methods in terms of stored solution displays.

The flexible and convenient *DARE CRT Editor program*[5,6] permits overwriting and correction, insertion of text, and automatic search for lines containing selected strings.

A SORT/EDIT program (precompiler) sorts the symbol string constituting the program and creates a FORTRAN differential-equation-solving program, which is then compiled and executed. After the first run, data such as system parameters and initial conditions may be changed on the CRT screen, and successive differential-equation solving-runs are obtained without recompilation. Iterative and statistical simulation studies can be programmed with FORTRAN statements.[5]

A new homemade graphic display[7] associated with our DARE console displays up to four variables against time, or selected phase-plane plots. The display uses one dual 9-bit (18-bit) word per display point to save memory and refresh time, can generate line segments for curve interpolation, and shares the processor memory through a standard PDP-9 data channel. This permits fast display refreshing with a minimum of time-wasting instructions.

## *DARE II: A fast block-macro system with an efficient precompiler*

The DARE I system demonstrates the convenience and power of a scale-factor-free, floating-point, equation-oriented, on-line simulation at relatively low computing speed. But *we also wanted to demonstrate a much faster on-line simulation system, which would permit true real-time flight simulation, still using the same small and inexpensive digital computer.* With the PDP-9, this meant giving up floating-point operation. DARE II machine equations must be scaled (much like those in analog computers) between $-1$ and 1 machine unit; with the PDP-9, ones-complement coding is employed. Overloads are detected and displayed by a special subroutine.

To provide high execution speed, DARE II uses the PDP-9 macro-assembler to create *macros corresponding to analog computing blocks*, an approach first used by Gaskill and McKnight in their batch-processed DAS system on the IBM 7090.[2] Our system permits especially convenient block programming, with each block named by type and by the actual output-variable name. The example of Figure 2 is represented by

```
SUM     F1, S1DOT, S2DOT
COS     COSA, A                    (1)
MULT    S1DOT, COSA, RDOT
```

*where the first argument of each block-macro represents the block output.* Note the convenient mnemonics used.

DARE II block-statements and data are entered on the dual-CRT console used also with DARE I and can be edited, modified, and printed out with the aid of the same string-processing editor.[6] DARE II simulations of many *small* systems (second to sixth order) are, however, so fast that *repetitive simulation and display* at two to 20 computer runs per second is possible. Keyboard entry of parameters is then too

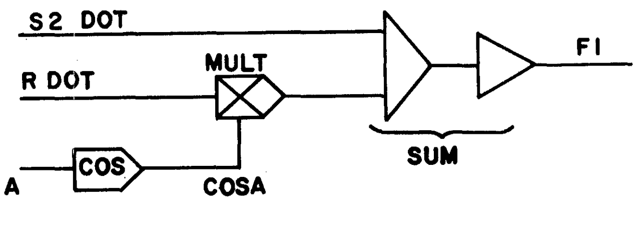


Figure 2—A block diagram

slow for CRT demonstration of parameter-change effects, and a "diddle knob" or joystick permitting rapid changes of a keyboard-addressed parameter will be added. The knob or joystick will control incrementation of an up-down counter holding the parameter value.

DARE II software incorporates substantial improvements over the DAS system. Block-macros may be typed in any order. An *optimizing precompiler* sorts statements like those in our example (1) before assembly, so that each block of the sorted program can operate on already computed quantities:

```
COS     COSA, A
MULT    SIDOT, COSA, RDOT        (2)
SUM     F1, SIDOT, S2DOT
```

This will then permit, say, integration of the output F1. *DARE II next employs conditional assembly*[10] *to completely eliminate the assembly of code for redundant store-fetch pairs corresponding to outputs and inputs of interconnected blocks.* Thus, the first macro COS COSA, A in (2) would ordinarily *end* with

```
STORE   COSA                     (3)
```

while the second macro MULT SIDOT, COSA, RDOT would *start* with

```
FETCH   COSA                     (4)
```

DARE II automatically cancels the redundant pair of instructions (3), (4), although (3) *would* be kept if it were needed elsewhere in the program. The pair

```
STORE SIDOT, FETCH SIDOT
```

will be similarly cancelled, unless SIDOT is needed elsewhere. *The DARE II precompiler program is specifically designed to permit elimination of as many track-store pairs as reasonably possible.* In addition, conditional assembly also eliminates code for unused multi-input-summer inputs and similar unused options. *As a result, DARE II produces code which is essentially as efficient as well-written PDP-9 machine-language code and permits relatively very fast execution* (Table II). If core storage is scarce, DARE II block macros can be subroutine calls to save core at the expense of some computing time.

Although the basic PDP-9 instruction set is quite limited (no byte manipulation, spare registers, or add-

TABLE II—Estimated computation times for a typical aerospace-vehicle simulation (TIMES are in μsec except as noted)

| OPERATION | NUMBER REQUIRED | DARE I PDP-9/FORTRAN (Floating-point) | DARE II PDP-9/Macro-assembler (Fixed-point) | DARE III/IV CDC 6400 (Floating-point) | 197X System (Floating-point) |
|---|---|---|---|---|---|
| X + Y + Z | 100 | ×1000 = 100,000 | ×5 = 500 | ×3.4 = 340 | ×0.2 = 20 |
| XY | 80 | ×700 = 56,000 | ×24 = 1920 | ×7 = 280 | ×1.2 = 96 |
| AX | 60 | ×700 = 42,000 | ×21 = 1260 | ×7 = 420 | ×1.2 = 72 |
| F(X) | 8 | ×4000 = 32,000 | ×52 = 416 | ×80 = 640 | ×10 = 80 |
| SINX or COS X | 10 | ×600 = 60,000 | ×60 = 6002 | ×100 = 1000 | ×15 = 150 |
| TOTAL—ONE DERIVATIVE EVALUATION | → | 290 msec | 4.7 msec | 2.7 msec | 0.46 msec |
| Two Derivative Evaluations | → | 580 msec | 9.4 msec | 5.4 msec | 0.9 msec |
| RK2 Integration | 12 | ×3000 = 36,000 | ×120 = 1440 | ×25 = 300 | ×4 = 48 |
| Total Frame Time DT | → | 616 msec | 11 msec | 5.7 msec | 1.4 msec |
| Max. Frequency at 25 Frames/cycle | → | 0.07 Hz | 4 Hz | 7 Hz | 30 Hz |

into-memory), many analog-computer blocks can be emulated quite nicely. As an example, a single-variable function with 256 uniformly spaced breakpoints can be formed by table lookup *and interpolation* in 50 $\mu$sec, and a two-variable function with $16 \times 16$ breakpoints can be formed in 120 $\mu$sec.[9] It is also readily possible to add to the DARE II macro-block repertoire; one can, for instance, create blocks which precisely correspond to the computing elements of any given analog computer.

Like DARE I, DARE II offers a choice of integration routines. Because PDP-9 lacks true index registers, the second-order Runge-Kutta routine[4]

$$\begin{aligned} {}^{k+1}X &= {}^{k}X + \tfrac{1}{2}(K_1 + K_2) \\ K_1 &= DT\,F({}^{k}X, k\,DT) \\ K_2 &= DT\,F[{}^{k}X + K_1, (k+1)\,DT] \end{aligned} \tag{5}$$

is probably the most useful, although it requires two evaluations of the derivative F(X, T) at each integration step. To implement Eq. (5), our program does not first evaluate all n $K_1$'s and then proceed to add half of each to its ${}^{k}X$, as might be done with a real index register. The program instead computes each ${}^{k}X + \frac{1}{2} K_1$ and ${}^{k}X + K_1$ before the next $K_1$ is evaluated. When this is finished for all X, the program sets a tally switch to mark the second part of the Runge-Kutta routine, increments the independent variable, and uses the ${}^{k}X + K_1$ to produce the $K_2$ and the ${}^{k+1}X$ as each derivative is computed. All integrand accumulation is done in double precision to reduce roundoff-error effects.

With suitable interrupts from a real-time clock, a DARE II simulation could be readily linked to a hybrid-computer setup and/or to real system hardware (autopilot, operator positions). Note, in this connection, that the macro-assembler system would circumvent the reentrancy problems usually encountered in attempts to service multiple system interrupts with FORTRAN programs.[3]

## *A look into the future: DARE III and DARE IV*

The DARE I and DARE II systems are expected to be completed in 1969. A useful and readily feasible next step could employ a modern 24 to 36 bit machine somewhat larger than our PDP-9 (e.g., SEL 840B, SDS Sigma 5, DEC PDP-10) to speed DARE I execution, or to add floating-point capability to DARE II. Such a system would cost between \$120,000 and \$200,000, which still matches the cost of a comparable analog-hybrid computer. Far more interesting from the point of view of economy as well as computing speed, however, is the possibility of *time-sharing* a substantially larger central digital computer, such as a CDC 6400. In fact, economical operation of even a medium-sized digital machine mainly intended for simulation should provide for time sharing with a "background" batch-processing program.

*Our proposals for follow-on projects, then, envisage implementation of DARE I- and DARE II-like simulation systems with the University's CDC 6400, using the existing PDP-9/console combination as a remote user's station.* 6400 activity would be restricted to very fast and efficient compiling and execution of differential-equation-solving programs, while the string-processing CRT editor, data entry and display, and also some iterative and statistics routines in slow simulations, would be performed by the small processor associated with the user's console. It is interesting to note that the simulation programs and data sent *to* the central computer involve only character strings transmitted at type-in rates. Alphanumerical data *from* the central computer do not require much higher rates; extensive numerical tables could be line-printed at the central installation. Each DARE CRT display, which is refreshed by the console processor, involves at most 2400 9-bit data samples. For typical 10 sec flight simulations, this would require transmission of 21,600 bits every 10 sec, or less than 2500 bits/second, so that a telephone line would do. Such operation is thus ideally suitable for remote time-sharing, provided that the 10-second-plus-overhead computer runs can be made available without excessive delays.

Based on initial DARE II experience, smaller simulation problems would be solved much more rapidly, say in 0.1 sec of central-processor time. Repetitive console displays demonstrating parameter-change effects would *not* be possible with reasonable data-transmission rates (nor would many such demonstrations be economically feasible)! Our proposed time-sharing scheme is, however, ideally suited to *fast iterative simulation or statistics-taking by the central processor* In this type of operation, only successive criterion-function values, accumulated statistics, or similar *numbers*, need to be transmitted and displayed during the iteration runs, and low transmission rates would again suffice.

In a console simulation system specifically designed for remote time sharing, our PDP-9 is really unnecessarily elaborate and could very effectively be replaced by the less costly 8K PDP-15, with DECtape but without extended arithmetic. Such a system, including very reasonable display facilities, would cost well under $50,000. An even less expensive system could be readily based on an even smaller 12- to 16-bit computer. This would save another $10,000; but the 18-bit word length of the PDP-15 is especially efficient for display-refreshing purposes and adds to the stand-alone capabilities of the console. Note, in this connection, that our own PDP-9-based console could employ DARE I for complete problem debugging before ever using CDC 6400 time.

With the large central computer and its relatively efficient compiler available, the proposed DARE III and DARE IV systems corresponding the the FORTRAN-based DARE I and the assembler-based DARE II, may well merge into each other. The multiple indexing needed for efficient implementation of integration routines may well be done best by the CDC 6400 FORTRAN compiler, while derivative computations would probably still be executed more efficiently by an assembler-based system employing conditional assembly, as in the DARE II scheme.

*Digital vs. analog/hybrid simulation: Computing-speed considerations*

Table II lists detailed estimates for various digital computation times required in a typical medium-sized aerospace simulation. Our example involves 12 state-variable-derivative integrations, 100 three-term additions, 140 products, and 18 functions of one variable. The DARE I and DARE II systems are implemented on a Digital Equipment Corporation PDP-9 (one μsec cycle time). This machine was chosen because it has an 18-bit rather than a 16-bit word length, although some of the newer 16-bit machines have much better instruction sets. The PDP-9 FORTRAN compiler appears to be designed mainly to save core storage and produces relatively very slow execution. At a reasonably conservative 25 frames (time increments DT) per period, the resulting 616-msec frame time for our aerospace simulation would permit the DARE I system to produce sinusoidal oscillations at 0.07 Hz. Speedwise, we see that the only differential analyzer our DARE I system replaces is an old-fashioned Bush or General Electric mechanical differential analyzer!

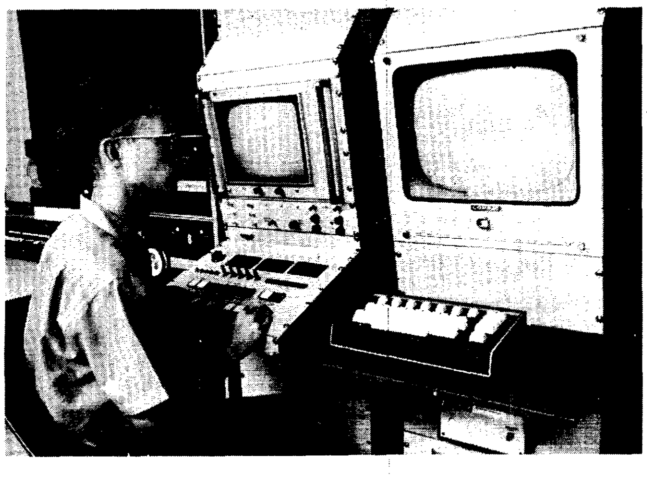


Figure 3—DARE console in operation with the PDP-9

A notable and inexpensive improvement in this situation is afforded by the fact that several PDP-9-sized digital computers are already available with hardware floating-point arithmetic. No such option is available with the PDP-9, but we ourselves have designed a current-mode logic, floating-point arithmetic unit for the PDP-9 which, if and when installed, would yield a speed improvement by a factor of at least 15 for the DARE I system, so that our simulated aerospace vehicle could wiggle at about 1 Hz, floating-point.

Our block-oriented DARE II system, also running on the PDP-9, was specifically designed to demonstrate relatively high-speed, real-time flight simulation on the inexpensive computer. The price paid for this is fixed-point operation, *but DARE II's efficient execution and 11-msec frame time permits about 4 Hz in the aerospace-simulation example.*

An improved 18- to 24-bit stand-alone computer of the future could probably produce comparable *floating-point* simulation at 4 Hz. As we have noted, though, the DARE III/IV systems will implement the economically much more important goal of time-shared operation with a large central digital computer, in this case the CDC 6400. As we have seen, very efficient and still relatively machine-independent execution will be obtained by FORTRAN integration and macro-assembler implementation of derivative computations, although many operators may prefer an entirely equation-oriented approach. In either case, *Table I indicates estimated frame times of the order of 5.7 msec, thus permitting about 7 Hz operation at 25 frames per cycle. Note that this system would provide floating-point aerospace-vehicle simulation in real time.*

The last column of Table II extrapolates the DARE III/IV system to a hypothetical 1970X digital computer permitting an approximately fivefold increase in computing speed through faster hardware and/or multiprocessing, instruction look-ahead, or hard-wired subroutines. *This is in no sense a way-out extrapolation, since digital-computer projects now on the drawing boards already plan for a fifty-fold speed increase.* Probably the most time to be gained in simulation calculations would be through the availability of fast scratch-pad memories or multiple registers, which would permit derivative computations with as few core-memory references as possible; this will already be approximated in the assembler version of our CDC-6400 simulation program. Additional computing bandwidth would readily be obtained with computer systems employing parallel multiple processors, which would fit nicely into differential-equation solving schemes. Note, however, that no manufacturer of large digital computers would even consider a special design for continuous system simulation, so that all improvements must make, as it were, incidental usage of developments in large-scale scientific and business computers.

Let us now consider the computing-speed situation on the analog/hybrid computer side. One or two analog computers available for sale in 1970 will offer not only 0.02 percent of half-scale static accuracy, but also 0.1 percent of half-scale error in linear computations at frequencies up to 1 KHz; multiplication and function generation are somewhat less accurate. In applications where such component accuracies suffice, even existing analog computers are thus seen to have a 20:1 speed advantage over the fastest digital-simulation systems. *This bandwidth advantage is moreover, not likely to decrease within the next ten years*; since 1965, improved $\pm$ 10-V hybrid computers developed in our laboratory have operated with errors below 0.2 percent for linear and one percent for nonlinear operations up to *10 KHz*, at perfectly reasonable cost.[11,12]

*Digital versus analog/hybrid: Economics*

Our DARE system is implemented on about \$90,000 worth of PDP-9 and simulation console; another \$25,000 could be very advantageously spent on a disk to speed compilation. When implementing the fixed-point DARE II language, our stand-alone system is roughly comparable to a modest 150-amplifier hybrid computer of 1960 vintage, say, an Electronic Associates 231-R together with a small digital computer used for potentiometer setup, static checking, and some function generation.

At a more or less comparable price, the on-line digital system is incomparably more convenient to program, check out, and operate (this is, of course, doubly true of the floating-point system). We also have, of course, all the possibilities of the 16K PDP-9 with dual display and can produce floating-point check solutions with DARE I.

Our PDP-9 installation is, however, mainly intended as a demonstration. A more useful stand-alone installation, based perhaps on the SDS Sigma 5, would roughly double our cost, but would permit real-time floating-point flight simulation, plus some foreground-background time sharing. Although such a system would be economically competitive with a 1970 analog/hybrid computer in many applications, *the full economic potential of on-line digital simulation will be realized only in a time-sharing system.* The tremendous advantage of the time-sharing system is, simply, that the central processor is free for other business while the simulation user looks at his console-refreshed display, or simply scratches himself. We have already seen that the communication requirements for time-shared simulation are quite small.

I believe that the foregoing considerations clearly indicate the area of future analog/hybrid vs. digital simulation competition. *In applications where analog/hybrid and digital simulation systems compete at equal computing speeds, i.e., in most real-time or "slow" simulation, the new digital systems will win overwhelmingly both on economic and on human-engineering grounds.* Since, on the other hand, reasonably complex nonlinear digital simulations will not be able to run at frequencies much in excess of 100 Hz, *faster simulation will still belong on analog/hybrid computers.*

A crucial question confronting the simulation community (and specifically the analog-computer industry) is, then, this: *where, and how large, are the application areas of really fast analog/hybrid computation?* The most immediately important would seem to be:

1. *Parameter and functional optimization*, including trajectory optimization.
2. *Random-process simulation*, including optimization of statistics, communication-system simulation, and parameter-tolerance studies.
3. *Solution of partial differential equations*, including techniques requiring multiplexing of analog computing elements.

It is in precisely these applications that the very large number of computer runs needed may give the analog/hybrid computer a measure of economic advantage even over digital batch processing. Even here, only important and frequent applications could tilt the balance away from time-shared digital simulation, which saves much analog-computer scaling, setup, checkout, and "head-scratching" time, not to speak of computer amortization. Cost estimates for different simulation methods sometimes omit these "hidden" costs.

I wonder, finally, how much practical high-speed analog/hybrid computation is really done in the aerospace, chemical and nuclear-energy industries, which are, at this time, the principal consumers of continuous-system simulation. Our own laboratory's work on the design and applications of very fast analog/hybrid computers,[11,12] for instance, has always elicited much polite interest, but very little imitation. By contrast, much current aerospace work involves "slow" or real-time hybrid simulation of aerospace systems, with the digital computer doing housekeeping functions such as static checking, plus function generation and, perhaps, some accurate trajectory integration. The resulting accuracy and software problems combine all the *worst* features of both analog and digital computation; the main reason for employing hybrid simulation at all is either the existence of actual hardware in the loop or some 20- to 50-Hz components due to hydraulic servos and/or aeroelasticity. This type of hybrid simulation can be swallowed by future on-line digital systems like Jonah by the whale. For the 1970s, the simulation community would be well advised to include on-time digital simulation in its planning, together with some careful reconsideration of faster analog/hybrid techniques.

## ACKNOWLEDGMENTS

The writer is grateful to the National Science Foundation for supporting Project DARE under NSF Grant GK 1860, and to Dr. R. Mattson, Head, Electrical Engineering Department, The University of Arizona for contributing University facilities. Project DARE software and hardware are being developed by a group of graduate students in the Electrical Engineering Department, including H. M. Aus, D. Chinnock, J. Goltz, T. Liebert, J. Puls, and A. Trevor. Professor J. V. Wait is co-principal investigator.

## REFERENCES

1 SCI SOFTWARE COMMITTEE
*The SCi continuous-system simulation language*
Simulation Dec 1967

2 R D BRENNAN R N LINEBARGER
*A survey of digital simulation*
Simulation Dec 1964

3 B JOHNSON
*Real-time digital simulation*
Proc IBM Symposium on Digital Simulation 1964

4 P R BENYON
*Review of numerical methods for digital simulation*
Simulation Nov 1968

5 J GOLTZ
*The DARE I on-line continuous-system simulation system*
ACL Memo 169 Electrical Engineering Dept
The Univ of Ariz 1969

6 *A PDP-9/Cathode-ray-typewriter editor*
ACL Memo 164 Electrical Engineering Dept The Univ of Ariz 1968

7 G A KORN *et al*
*A new graphic display/plotter for small digital computers*
Proc SJCC 1969

8 A TREVOR J V WAIT
*DIFFE: An on-line differential-equation solving routine with automatically scaled display*
ACL MEMO 153 Electrical Engineering Dept the Univ of Ariz 1968

9 H M AUS G A KORN
*Table-lookup/interpolation function generation for fixed-point digital computations*
IEEETEC August 1969

10 M D McILROY
*Macro-instruction extensions of compiler languages*
C ACM April 1960

11 G A KORN
*Progress of analog/hybrid computation*
Proc IEEE Dec 1966

12 B K CONANT
*A new solid-state iterative differential analyzer making maximum use of intergrated circuits, Proc.* FJCC *1968.*

# MOBSSL-UAF—An augmented block structured continuous system simulation language for digital and hybrid computers

*by* M. J. MERRITT and D. S. MILLER

*USC School of Engineering*
Los Angeles, California

## INTRODUCTION

The motivation for the development of digital simulation languages may be seen by tracing the thoughts of two widely different people preparing to analyze a continuous dynamic system. Both are experienced engineers and mathematicians, but the first is a novice programmer with little or no FORTRAN experience. Both have access to one or more digital computers. The novice's thoughts might be as follows: "I do not know FORTRAN and I'm not really interested in learning it just to solve this problem. I have heard that digital continuous simulation languages are simple and easy to use. I'll try one". The experienced programmer, on the other hand, might think, "I only need a few quick solutions, why bother with a FORTRAN program. I'll use a simulation language for convenience."

Clancy and Fineberg,[1] in 1965, compiled a comprehensive list of some 31 simulation languages. One of these would fit the needs as well as the computer of both individuals. The novice is looking for a simple easy to use language. The experienced programmer is looking for one that compiles and runs efficiently while providing as much flexibility and convenience as possible. Since none of the presently available languages achieve the same running efficiency as a FORTRAN program written specifically to solve the same problem its conveniences must weigh heavily in the programmer's mind.

If a language is to satisfy the needs of these, as well as a broad spectrum of users in between, then it must possess the following characteristics:

1. It must be easy to learn.
2. Its language statements must be simple and easy to interpret.
3. It should not require any knowledge of FORTRAN.
4. It should allow on-line interaction during both problem preparation and problem execution.
5. The language should contain sufficient computational control, and input/output elements so that only exceptionally complex tasks require FORTRAN or other non-simulation language statements.

Of the widely distributed languages, PACTOLUS[2] and IBM 1130 CSMP[3] come the closest to meeting these requirements. Unfortunately, they lack many necessary computational and control functions. The popular MIDAS[4] language is not interactive, while MIMIC, DSL 90 and 360 CSMP[5] are difficult to learn and very FORTRAN oriented.

All of these requirements may be met by combining two things: a computer graphics terminal, and an augmented block structured simulation language. The graphics terminal for its interactive communication abilities and the block structured simulation language because of its simple language statements. Further, the graphics terminals ability to display large quantities of instructional and reference information quickly, allows it to guide the new programmer through each step of the problem preparation.

A block structured language may be visualized as a collection of input-output boxes (see Table 1), each of which carries out a basic mathematical operation. The user's inputs, the language statements, describe the way in which these pre-defined functional blocks are to be inter-connected. A typical language statement might be: 54, M, 1, 7 which might mean: the output of the block element designated as #54 is the product of the outputs of the block elements designated #1 and #7. The advantage of block structured language (MIDAS PACTOLUS, 1130 CSMP) lie in the simplicit of their language statements .Their major disadvantage is their rigidity, i.e., the user is restricted to those operations which may be mechanized with the available mathematical and control operations. This disadvantage may be overcome by constructing process oriented block elements which cause higher order mathematical operations to be carried out. The Gradient Processor and Disk Input/Output block elements, described below, are two such elements.

## *The MOBSSL language*

MOBSSL-UAF, which stands for *M*erritt and Miller's *O*wn *B*lock *S*tructured *S*timulation *L*anguage—*U*npronounceable *A*cronym *F*or, is a descendent of MIDAS through PACTOLUS and IBM 1130 CSMP It differs from its antecedents in the following ways:

1. Continuous and iterative gradient modeling and optimization procedures are performed by a Gradient Processor block element.
2. Analog to Digital and Digital to Analog conversion block elements facilitate closed loop hybrid computation, On-line interaction and control of analog plotting devices: x-y plotters, stripchart recorders, memoscopes and oscilloscopes.
3. A Disk output block element allows up to 10 block outputs to be written in a pre-defined disk data set. A Disk Input block element reads up to 10 inputs from a pre-defined data set. Utility subroutines allow these data sets to be referenced by FORTRAN programs.
4. Iterative and parametric computations are facilitated by allowing control cards to specify a SIMULATION MODE. When a solution is completed, the SIMULATION MODE determines which of the following is to occur:

   STOP—terminates the job.

   PCHG—read data cards and modify parameters and initial conditions accordingly. The last data card specifies the SIMULATION MODE for the next solution which is begun immediately.

   RUN—begin a new solution immediately. Successive solutions may be modified by on-line control or the gradient and iterative block elements. This mode contains no exit and must be terminated by operator intervention or by forcing an error exit, i.e., take the square root of a negative number.

Process oriented block elements, like the Gradient Processor and the Disk Input/Output blocks, make it possible for unsophisticated programmers to study complex dynamic systems, modeling and optimization problems, and exercise on-line control without first learning the FORTRAN language.

The communication and interactive features of MOBSSL, the Hybrid block elements and the SIMULATION MODE, were dictated by the computational facilities of the System Simulation Laboratory at the University of Southern California. In this laboratory, each user receives ten minutes of computer time on a first come, first served *programmer present* basis. This period is too short to encourage the use of the console typewriter for communication purposes. Instead, the user may read pre-planned parameter changes from punched cards, or operate control switches and potentiometers connected to the Hybrid block elements. The effects of these changes are observed in the line printer listings or on analog displays operated by the Hybrid elements.

## *The gradient processor*

Optimization and modeling of synamic dystems may be re-formulated as a search for the extrema of a scalar functional of a vector with free parameters.

TABLE I—Definition of MOBSSL elements

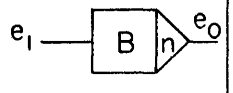
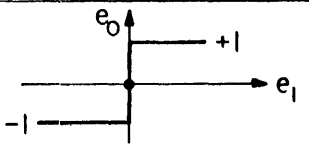
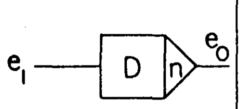
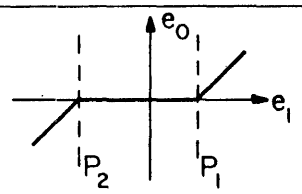
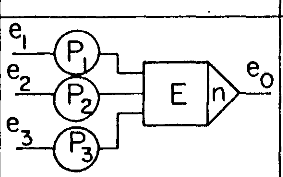
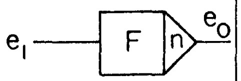
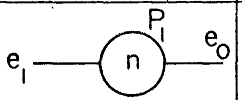
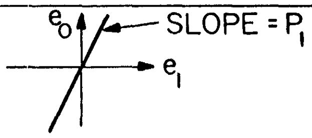
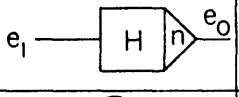
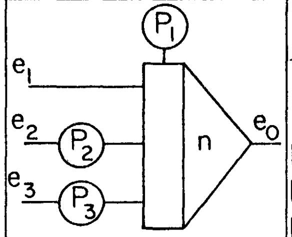
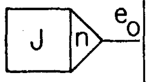
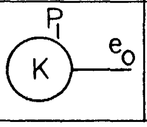
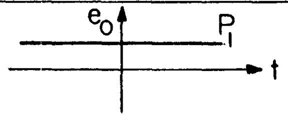
| ELEMENT TYPE | MOBSSL TYPE CODE | BLOCK DIAGRAM SYMBOL | DESCRIPTION & COMMENTS |
|---|---|---|---|
| BANG-BANG | B |  |  $e_0=-1$ $-\infty<e_1<0$ <br> $e_0=0$ $e_1=0$ <br> $e_0=+1$ $0<e_1<+\infty$ |
| DEAD SPACE | D |  |  $e_0=e_1-P_2$ $-\infty<e_1\le P_2$ <br> $e_0=0$ $P_2<e_1<P_1$ <br> $e_0=e_1-P_1$ $P_1\le e_1<+\infty$ <br> $P_1\ge 0$ $P_2\le 0$ |
| EXPONENT | E |  | $e_0=e^{(P_1e_1+P_2e_2+P_3e_3)}$ |
| * ~~FUNCTION~~ GENERATOR | F |  | $e_0=F(e_1)$ |
| GAIN | G |  |  $e_0=P_1e_1$ |
| HALF POWER | H |  | $e_0=\sqrt{e_1}$ $e_1>0$ |
| * INTEGRATOR | I |  | $e_0=P_1+\int_0^t(e_1+P_2e_2+P_3e_3)\,dt$ <br> $e_0(t_0)=P_1$ <br> Maximum no. of Integrators = 75 <br> Minimum no. of Integrators = 1 <br> Fourth order Runge-Kutta |
| * JITTER | J |  | Random Number Generator <br> Generates random number between ±1 <br> $-1\le e_0\le +1$ |
| CONSTANT | K |  |  $e_0=P_1$ |

TABLE I—Definition of MOBSSL elements

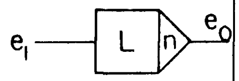
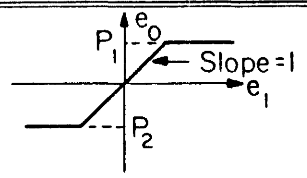
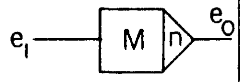
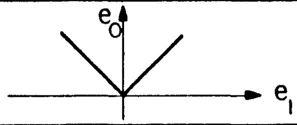
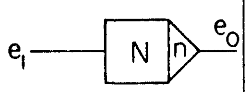
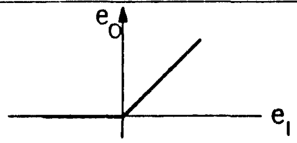
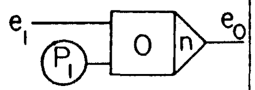
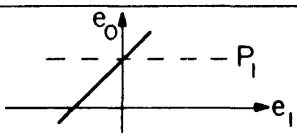
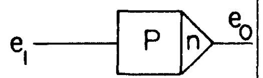
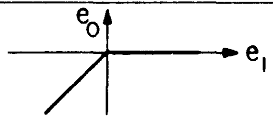
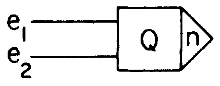
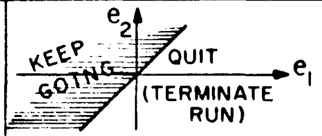
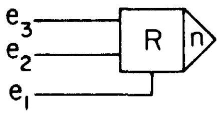
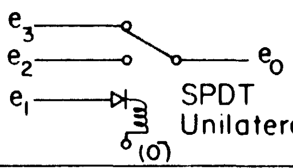
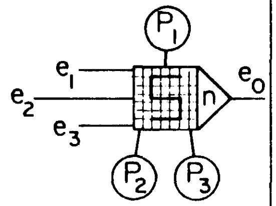
| ELEMENT TYPE | MOBSSL TYPE CODE | BLOCK DIAGRAM SYMBOL | DESCRIPTION & COMMENTS |
|---|---|---|---|
| LIMITER | L |  |  $e_0 = P_2 \quad -\infty < e_1 < P_2$<br>$e_0 = e_1 \quad P_2 \leq e_1 < P_1$<br>$e_0 = P_1 \quad P_1 \leq e_1 < +\infty$<br>$P_1 \geq P_2$ |
| MAGNITUDE | M |  |  $e_0 = \lvert e_1 \rvert$ |
| NEGATIVE CLIPPER | N |  |  $e_0 = 0 \quad -\infty < e_1 \leq 0$<br>$e_0 = e_1 \quad 0 < e_1 < +\infty$ |
| OFFSET | O |  |  $e_0 = e_1 + P_1$ |
| POSITIVE CLIPPER | P |  |  $e_0 = e_1 \quad -\infty < e_1 < 0$<br>$e_0 = 0 \quad 0 \leq e_1 < +\infty$ |
| QUIT | Q |  |  $e_1 > e_2 \Longrightarrow$ QUIT<br>$e_1 \leq e_2 \not\Longrightarrow$ QUIT<br>Quit element terminates the run at the end of the DT step in progress |
| RELAY | R |  |   $e_3$<br>$e_2$<br>$e_1$<br>$e_0$<br>(O)<br>SPDT<br>Unilateral Relay $e_0 = e_3 \quad -\infty < e_1 < 0$<br>$e_0 = e_2 \quad 0 \leq e_1 < +\infty$ |
| *STORAGE | S |   $P_1$<br>$e_1$<br>$e_2$<br>$e_3$<br>S<br>n<br>$e_0$<br>$P_2$<br>$P_3$ | Enables block outputs from end (TF) of a given run (determined by $P_2$) to be stored, that is, continually available for the succeeding $P_3$ runs<br>$e_0(\text{run } 1) = P_1$<br>$e_0[\text{run}(n+1)] = e_1[\text{run}(n)]_{TF} + e_2[\text{run}(n)]_{TF} + e_3[\text{run}(n)]_{TF}$ if $(n-P_2) \text{ MOD } P_3 = 0$<br>$e_0[\text{run}(n+1)] = e_0[\text{run}(n)]_{TF}$ if $(n-P_2) \text{ MOD } P_3 > 0$ |

TABLE I—Definition of MOBSSL elements

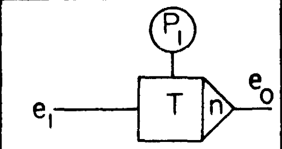
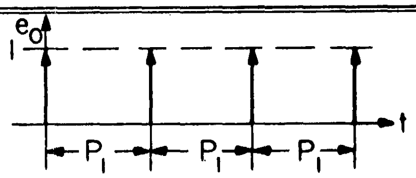
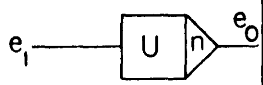
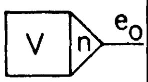
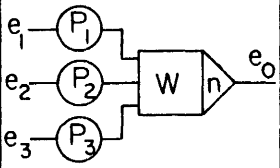
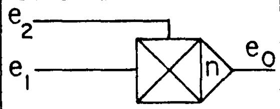
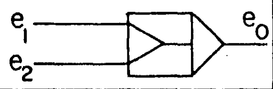
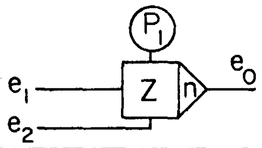
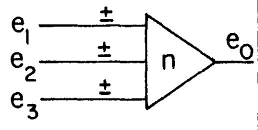
| ELEMENT TYPE | MOBSSL TYPE CODE | BLOCK DIAGRAM SYMBOL | DESCRIPTION & COMMENTS |
|---|---|---|---|
| * TIME PULSE GENERATOR | T |  |  Generates impulse train of unit amplitude and period $P_1$ which starts when $e_1 \geq 0$ (to delay start of pulse train keep $e_1$ negative). |
| * UNIT DELAY | U |  | $e_0(t) = P_1$ $t=0$ Max. no. of unit delays = 75<br>$e_0(t) = e_0(t-\Delta t)$ $t>0$<br>Used as a delay element and in conjunction with Z element for sampled data systems and difference equation computations |
| * VACUOUS | V |  | $e_0 = P_1$ $t=0$<br>Used in conjunction with WYE element for implicit function generation |
| WEIGHTED SUMMER | W |  | $e_0 = P_1e_1 + P_2e_2 + P_3e_3$ |
| MULTIPLIER | X |  | $e_0 = e_1e_2$ |
| * WYE | Y |  | Logical branch element used in implicit function generation |
| * ZERO-ORDER HOLD | Z |  | $e_0 = P_1$ if $t=0$ and $e_2 \leq 0$<br>$e_0 = e_1$ $e_2 > 0$<br>$e_0$ unchanged $e_2 \leq 0$ |
| SUMMER | + |  | $e_0 = \pm e_1 \pm e_2 \pm e_3$<br>This is the only element that accepts negative block numbers. |

TABLE I—Definition of MOBSSL elements

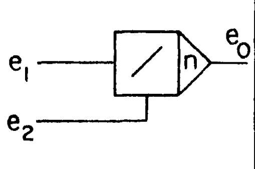
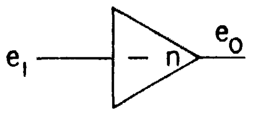
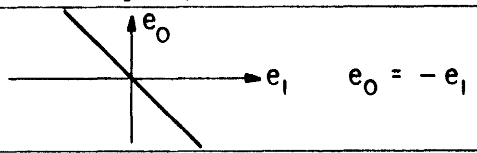
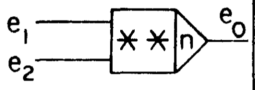
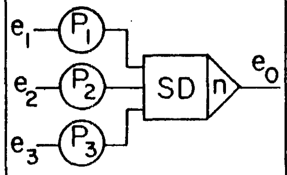
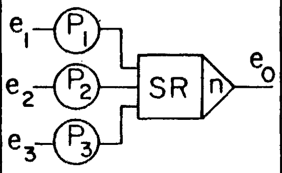
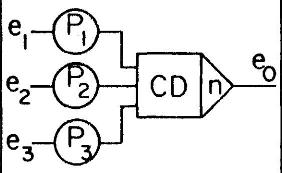
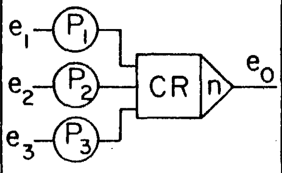
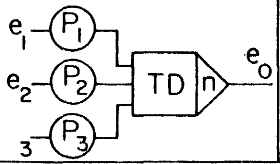
| ELEMENT TYPE | MOBSSL TYPE CODE | BLOCK DIAGRAM SYMBOL | DESCRIPTION & COMMENTS |
|---|---|---|---|
| DIVIDER | |  | $e_0 = \frac{e_1}{e_2}$ $e_2 \neq 0$ If $e_2 = 0$, program interrupt occurs and 360 supervisor generates message indicating exponent overflow exception |
| INVERTER | — |  |  $e_0 = -e_1$ |
| POWER | ** |  | $e_0 = (e_1)^{e_2}$ $e_1 > 0$ If $e_1 \leq 0$, problem processing is terminated and 360 supervisor generates error message indicating an attempt to take logarithm of a number $\leq 0$ has occurred |
| SINE (DEGREES) | SD |  | $e_0 = \text{SIN}(P_1e_1 + P_2e_2 + P_3e_3)$ Inputs in degrees |
| SINE (RADIANS) | SR |  | $e_0 = \text{SIN}(P_1e_1 + P_2e_2 + P_3e_3)$ Inputs in radians |
| COSINE (DEGREES) | CD |  | $e_0 = \text{COS}(P_1e_1 + P_2e_2 + P_3e_3)$ Inputs in degrees |
| COSINE (RADIANS) | CR |  | $e_0 = \text{COS}(P_1e_1 + P_2e_2 + P_3e_3)$ Inputs in radians |
| TANGENT (DEGREES) | TD |  | $e_0 = \text{TAN}(P_1e_1 + P_2e_2 + P_3e_3)$ Inputs in degrees |

TABLE I—Definition of MOBSSL elements

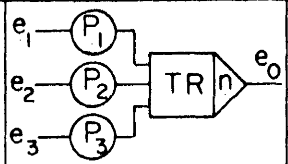
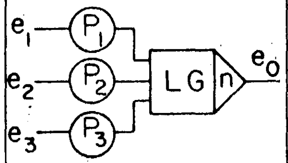
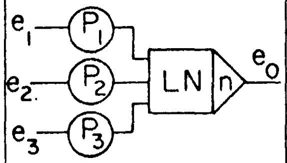
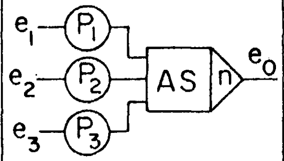
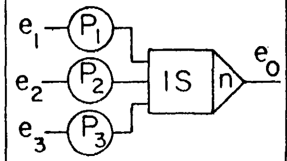
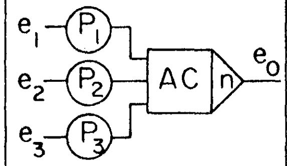
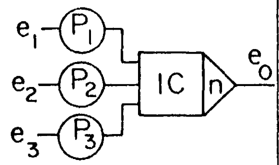
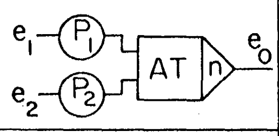
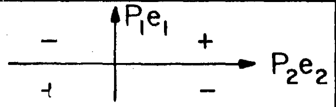
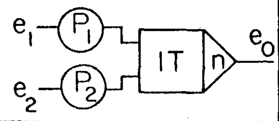
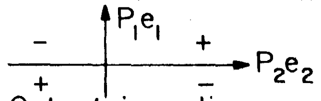
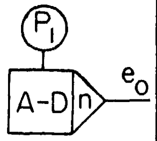
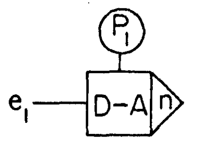
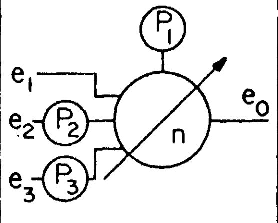
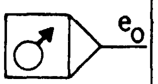
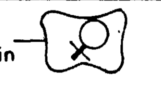
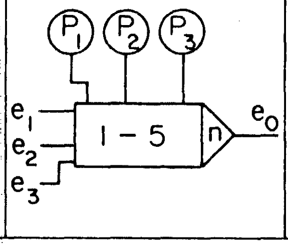
| ELEMENT TYPE | MOBSSL TYPE CODE | BLOCK DIAGRAM SYMBOL | DESCRIPTION & COMMENTS |
|---|---|---|---|
| TANGENT (RADIANS) | TR |  | $e_0 = \text{TAN}(P_1e_1 + P_2e_2 + P_3e_3)$ Inputs in radians |
| COMMON LOGARITHM | LG |  | $e_0 = \text{LOG}_{10}(P_1e_1 + P_2e_2 + P_3e_3)$ Base 10 |
| NATURAL LOGARITHM | LN |  | $e_0 = \text{LOG}_\epsilon(P_1e_1 + P_2e_2 + P_3e_3)$ Base $\epsilon$ |
| ARCSINE (DEGREES) | AS |  | $e_0 = \text{SIN}^{-1}(P_1e_1 + P_2e_2 + P_3e_3)$ Output in degrees |
| ARCSINE (RADIANS) | IS |  | $e_0 = \text{SIN}^{-1}(P_1e_1 + P_2e_2 + P_3e_3)$ Output in radians |
| ARCCOSINE (DEGREES) | AC |  | $e_0 = \text{COS}^{-1}(P_1e_1 + P_2e_2 + P_3e_3)$ Output in degrees |
| ARCCOSINE (RADIANS) | IC |  | $e_0 = \text{COS}^{-1}(P_1e_1 + P_2e_2 + P_3e_3)$ Output in radians |
| ARCTANGENT (DEGREES) | AT |  | $e_0 = \text{TAN}^{-1}\frac{P_1e_1}{P_2e_2}$  4 Quadrant operation, Output in degrees |
| ARCTANGENT (RADIANS) | IT |  | $e_0 = \text{TAN}^{-1}\frac{P_1e_1}{P_2e_2}$  4 Quadrant operation, Output in radians |

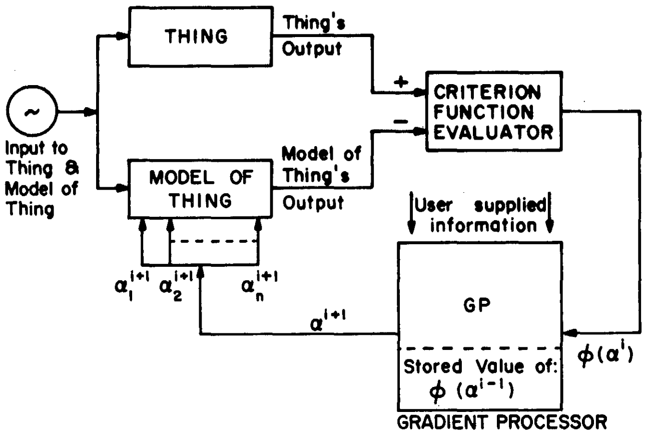
TABLE I—Definition of MOBSSL elements

| ELEMENT TYPE | MOBSSL TYPE CODE | BLOCK DIAGRAM SYMBOL | DESCRIPTION & COMMENTS |
|---|---|---|---|
| ANALOG to DIGITAL CONVERTER | AD |  | $e_0 = e_{ADC*P_1}$ $e_{ADC*P_1} < 100.0V$<br>Input at Beckman 2132 patchboard ADC $*P_1$ |
| DIGITAL to ANALOG CONVERTER | DA |  | $e_{DAC*P_1} = e_1$ $e_1 < 100.0$<br>Output at Beckman 2132 patchboard DAC $*P_1$ output. Apply + 100.0V to DAC $*P_1$ hi and − 100.0V to DAC$*P_1$ lo terminals. |
| * VARIABLE CONSTANT | VK |  | Enables user to vary constants from run to run. Output for run 1 is $P_1$. Output for subsequent runs is determined by output for previous runs and inputs at end of previous run (TF) and parameters $P_2$ and $P_3$<br>$e_0(\text{run } 1) = P_1$<br>$e_0[\text{run}(n+1)] = e_0[\text{run}(n)] + e_1[\text{run}(n)]_{TF} + P_2e_2[\text{run}(n)]_{TF} + P_3e_3[\text{run}(n)]_{TF}$ |
| MALE | ♂ |  | Used in conjunction with ♀ block to produce osculations.<br>For additional details see page 7430 |
| FEMALE | ♀ |  | Accepts only ♂ block as input.<br>Produces more blocks. |
| * SPECIAL ELEMENTS | 1<br>2<br>3<br>4<br>5 |  | User supplied Fortran subroutines not restricted to 3 inputs, 3 parameters and 1 output. Inputs and outputs of all blocks + all MOBSSL variables (T, ΔT, TTOT, TSAMP, etc.) are available. Approximately 1100 words of core available for this purpose. |

Consider the modeling problem shown in Figure 1. The task is to select those values of the parameter vector, $\alpha$ ,which result in minimizing the output of the Criterion Function Evaluator. The integral squared difference between the model output and the output of the unknown system is often selected as the scalar criterion function. The Gradient Processor, GP, block element controls the systematic variation of the parameter vector, $\alpha$, so as to locate the desired minima.

Let the criterion function or cost function which is to be extremized be denoted as $\phi(\alpha_1,\ldots,\alpha_n)$ or just $\phi(\alpha)$. If $\phi$ is a non-linear function of the parameter vector, $\alpha$, as it usually is, then iterative search procedures must be employed to find the extrema. Of the procedures described by Bekey and Karplus,[6] the most often used is the method of steepest ascent. The Gradient Processor, GP, block element mechanizes an iterative form of the method of steepest ascent described below.

The gradient of the criterion function on the ith



THING
Thing's Output
Input to Thing & Model of Thing
MODEL OF THING
Model of Thing's Output
CRITERION FUNCTION EVALUATOR
+
−
User supplied information
GP
Stored Value of: $\phi(a^{i-1})$
$\phi(a^i)$
$a_1^{i+1}$ $a_2^{i+1}$ $a_n^{i+1}$
$a^{i+1}$
GRADIENT PROCESSOR

Figure 1—Application of the gradient processor element to parameter identification

iteration $\Delta\phi^i$, in the $\phi x\alpha_1 x, \ldots, x\alpha_n$ space may be estimated by perturbing each parameter $\alpha_j$ by an amount $\Delta\alpha^+{}_j$ and $\Delta\alpha^-{}_j$:

$$\nabla\phi^i = \begin{bmatrix} \dfrac{\phi(\alpha_1{}^i + \Delta\alpha_1{}^+, \alpha_2{}^i, \cdots, \alpha_n{}^i) - \phi(\alpha_1{}^i - \Delta\alpha_1{}^-, \alpha_2{}^i, \cdots, \alpha_n{}^i)}{(\Delta\alpha_1{}^+ + \Delta\alpha_1{}^-)} \\ \vdots \\ \dfrac{\phi(\alpha_1{}^i, \alpha_2{}^i, \cdots, \alpha_n{}^i + \Delta\alpha_n{}^+) - \phi(\alpha_1{}^i, \alpha_2{}^i, \cdots, \alpha_n{}^i - \Delta\alpha_n{}^-)}{(\Delta\alpha_n{}^+ + \Delta\alpha_n{}^-)} \end{bmatrix}$$

This computation requires 2n solutions of the equations which determine $\phi$, with appropriate cyclic control of the parameter vector $\alpha$. At the conclusion of the 2n solutions, the gradient vector, $\nabla\phi$, is computed.

Let the ith estimate of the parameter vector be denoted by $\alpha^i$. Let $\alpha^0$ be any arbitrarily selected set of parameters, then the successive estimates of the $\alpha$ are computed from

$$\alpha^{i+1} = \alpha^i + M^i\nabla\phi^i$$

where $M^i$ is an n by n diagonal matrix of the form

$$\begin{bmatrix} m_1{}^i & & & & & \\ & m_2{}^i & & & 0 & \\ & & m_3{}^i & & & \\ & & & \cdot & & \\ & & & & \cdot & \\ & & & & & \cdot \\ 0 & & & & \cdot & \\ & & & & & m_n{}^i \end{bmatrix}$$

The $m_j$ are positive if a maximum of $\phi$ is sought (steepest ascent) and they are negative if a minimum is sought (steepest descent). The magnitudes of the $m_j$ may be used to restrict the size of each parameter step as follows:

Let $||\Delta\alpha^i||_u$, the unnormalized parameter step, be defined as

$$||\Delta\alpha^i||_u = ||\alpha^{i+1} - \alpha^i||_u \triangleq \sqrt{\sum_{n=1}^{n}\left[k\,\frac{\partial\phi(\alpha_n{}^i)}{\partial\alpha_n}\right]^2}$$

Let MSL be a pre-defined constant, equal to the largest parameter step, $\Delta\alpha$, desired.

If

$$||\Delta\alpha^i||_u > \text{MSL}$$

then

$$m_j{}^i = \frac{\text{MSL}\cdot k_j}{||\Delta\alpha^i||_u}$$

otherwise

$$m_j{}^i = k_j$$

where the $k_j$ are constants supplied as inputs to the GP blocks.

As with all iterative procedures, it is difficult to determine when to stop the iteration. The Gradient Processor block element offers three separate stopping options, all controlled by input parameters:

1. if the number of iterative cycles exceeds a specified number the simulation is terminated.
2. if $\phi$ is being maximized and exceeds a given value, or $\phi$ is being minimized and is less than a given value then the simulation is terminated.
3. If $|\phi(\alpha^{i+1}) — \phi(\alpha^i)|$ is less than a given constant, the simulation is terminated.

At the conclusion of each iterative cycle, a total of 2n + 1 runs, the values of the new parameter vector, $\alpha^{i+1}$ the old criterion function, $\phi(\alpha^i)$ and the new criterion function $\phi(\alpha^{i+1})$ and the magnitude of the stopping criteria being used are printed. Additional listings of the gradient vector, and individual parameter changes both before and after normalization are optional.

MOBSSL will accept up to 11 GP blocks. They must all be assigned sequential block numbers. The

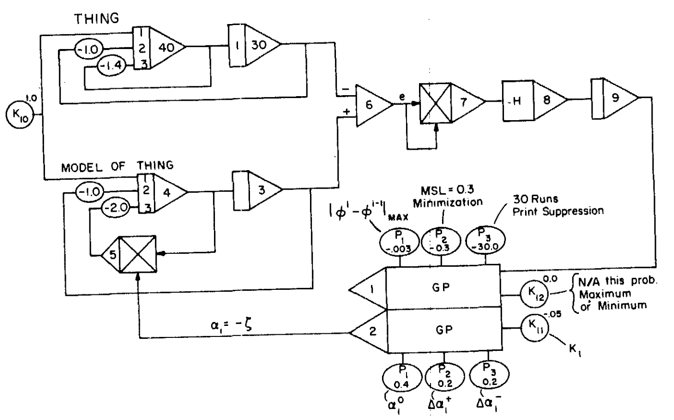


Figure 2—MOBSSL block diagram for second-order system damping ratio example

first GP block is not associated with a parameter, but accepts inputs and constants used to control the sequencing of the remaining GP blocks. The outputs of the last n GP blocks are the values of the n parameters $\alpha_1, \ldots, \alpha_n$ where $n \leq 10$. A single parameter modeling problem is shown in Figure 2.

The functions of the GP blocks inputs and parameters are given in Table II.

## Parameter identification using the GP element

The damping ratio, $\zeta$, of a linear, second order system is not known. The response of this second order system to a step input is available. A model equation is

$$\ddot{x} + 2\alpha_1\dot{x} + x = 1.$$

The actual damping ratio, $\zeta$, of the system was set to 0.7.

The MOBSSL program to carry out the iterative steepest descent minimization of the integral of the absolute value of the difference between the two step responses is shown in Figures 2 and 3. The initial value of the parameter $\alpha_1$ is selected as 0.4. The MOBSSL results for the first iteration are shown in Figure 4. The first column of tabular data shown is time, the second is the output of the 2nd GP block, $\alpha_1$, the 3rd column is the output of the Criterion Function Evaluator, which is $\phi(\alpha)$ at the end of a solution, the 4th column is the step response of the system containing the unknown parameter, and the last column and plot show the step response of the model containing parameter $\alpha_1$.

As can be seen from these results, $\alpha_1$, started at 0.4 and after one iteration had reached 0.6455, heading towards 0.7.

TABLE II—Gradient processor inputs and parameters

| | |
|---|---|
| FIRST GP BLOCK | |
| INPUT 1 | The Criterion Function, $\phi$. |
| INPUT 2 | A stopping criteria: Maximum or Minimum value of $\phi$ desired; usually supplied by a constant block. |
| INPUT 3 | Not used. |
| PARAMETER 1 | A Stopping Criteria: if $\|\phi(\alpha^{i+1}) - \phi(\alpha^i)\| \leq$ PAR 1, stop. If zero, using another criterion |
| PARAMETER 2 | If positive, maximize $\phi$. If negative, minimize $\phi$. Magnitude is largest allowable parameter step $\|\|\Delta\alpha\|\|$. |
| PARAMETER 3 | A stopping criteria: number of allowable iterations. If positive, print optional information. If negative, suppress it. |
| ALL OTHER BLOCKS | |
| INPUT 1 | Steepest ascent gain constant $k_j$ — usually supplied by constant block element. |
| INPUT 2, INPUT 3 | Not used. |
| PARAMETER 1 | Initial estimate of parameter value, $\alpha_j^0$. |
| PARAMETER 2 | Positive parameter perturbation, $\Delta\alpha_j^+$ |
| PARAMETER 3 | Negative parameter perturbation, $\Delta\alpha_j^-$. |

### *Iterative computational elements*

Iterative computational processes are facilitated by two MOBSSL elements: the STORAGE element and the VARIABLE CONSTANT element, designated S and VK respectively. These elements allow the results obtained in previous solutions to modify future solutions. When MOBSSL ends a solution, it examines the SIMULATION MODE established by the programmer's control cards. If the RUN mode is in effect, and the Gradient Processer, GP, element is not in use, the STORAGE and VARIABLE CONSTANT elements are processed to determine their new outputs. All other elements are reset to their initial conditions and the independent variable is reset to zero. The solution counter, N, which begins at 1, is incremented by 1. When all of the bookkeeping is completed, MOBSSL begins the new solution.

```
MOBSSL,UAF-- MERRITT'S OWN BLOCK STRUCTURED SIMULATION LANGUAGE, UNPRONOUNCEABLE ACRONYM FOR...MK II MOD 2    JAN 01 1969

CONFIGURATION SPECIFICATIONS

OUTPUT NAME        BLOCK NUMBER   BLOCK TYPE   INPUT 1   INPUT 2   INPUT 3
GP HEADER                1            GP           9        12         0
GP PARAM 1               2            GP          11         0         0
MODEL OF THING           3            I            4         0         0
MODEL OF THG DOT         4            I           10         3         5
PARAM MULTIPLIER         5            X            4         2         0
ERROR                    6            +           30         3         0
ERROR**2                 7            X            6         6         0
SQRT(ERROR**2)           8            H            7         0         0
CRITERION FCN            9            I            8         0         0
INPUT                   10            K            0         0         0
GRADIENT GAIN           11            K            0         0         0
GAIN FOR MIN/MAX        12            K            0         0         0
THING                   30            I           40         0         0
THING DOT               40            I           10        30        40

                       INITIAL CONDITIONS AND PARAMETERS

IC/PAR NAME      BLOCK      IC/PAR1           PAR2           PAR3
                    1         0.00300   -0.30000  -30.00000
                    2         0.40000    0.20000    0.20000
                   10         1.00000    0.0        0.0
                   11        -0.05000    0.0        0.0
                   12         0.0        0.0        0.0
                    4         0.0       -1.00000   -2.00000
                   40         0.0       -1.00000   -1.40000

                       PROGRAM MODE      STOP
INTEGRATION INTERVAL IS              0.04000
TOTAL TIME IS              10.00000
PRINT INTERVAL IS              0.40000
BLOCKS TO BE PRINTED ARE          2           9           30

BLOCK TO BE PLOTTED  IS          3 RANGE OF PLOTTED VARIABLE IS            0.0                 2.00000
```

Figure 3—MOBSSL listing of configuration, parameter and timing data for damping ratio example

## The VARIABLE CONSTANT element

The VARIABLE CONSTANT, VK, element is programmed in the same manner as the CONSTANT, K, element. In both cases, the element's output remains constant during a solution. The constant stored by the VK element is recomputed between successive solutions. Consequently, the VK element utilizes only the terminal values of its possibly time varying inputs. If information available interior to a solution is needed to modify the next solution, it must be stored in a sample and hold element until the end of the solution, at which time it may be used by a VK element.

The VK element presents a constant output for an entire solution, equal to its output on the previous run plus the sum of its first input, P2 times its second input, and P3 times its third input all at the end of the previous solution. The constants P1, P2 and P3 are the block element's three parameters. Its output is set equal to P1 during the first solution. The VK element is similar to a mechanical ratchet or to an accumulator.

The VK element is equivalent to an analog computer iterative accumulator. It allows solution to solution parameter variations These may be systematic changes, random changes or solution dependent changes. For example a frequency response can be implemented as shown in Figure 5. The VK block causes the radial frequency $\omega$ to be incremented by k at the end of each solution.

Consider the following parameter identification problems. Let

$$\dot{y}_D + \alpha\, y_D = 1$$

be a first order system containing an unknown parameter $\alpha$. Let

$$\dot{y} + \alpha_1 y = 1$$

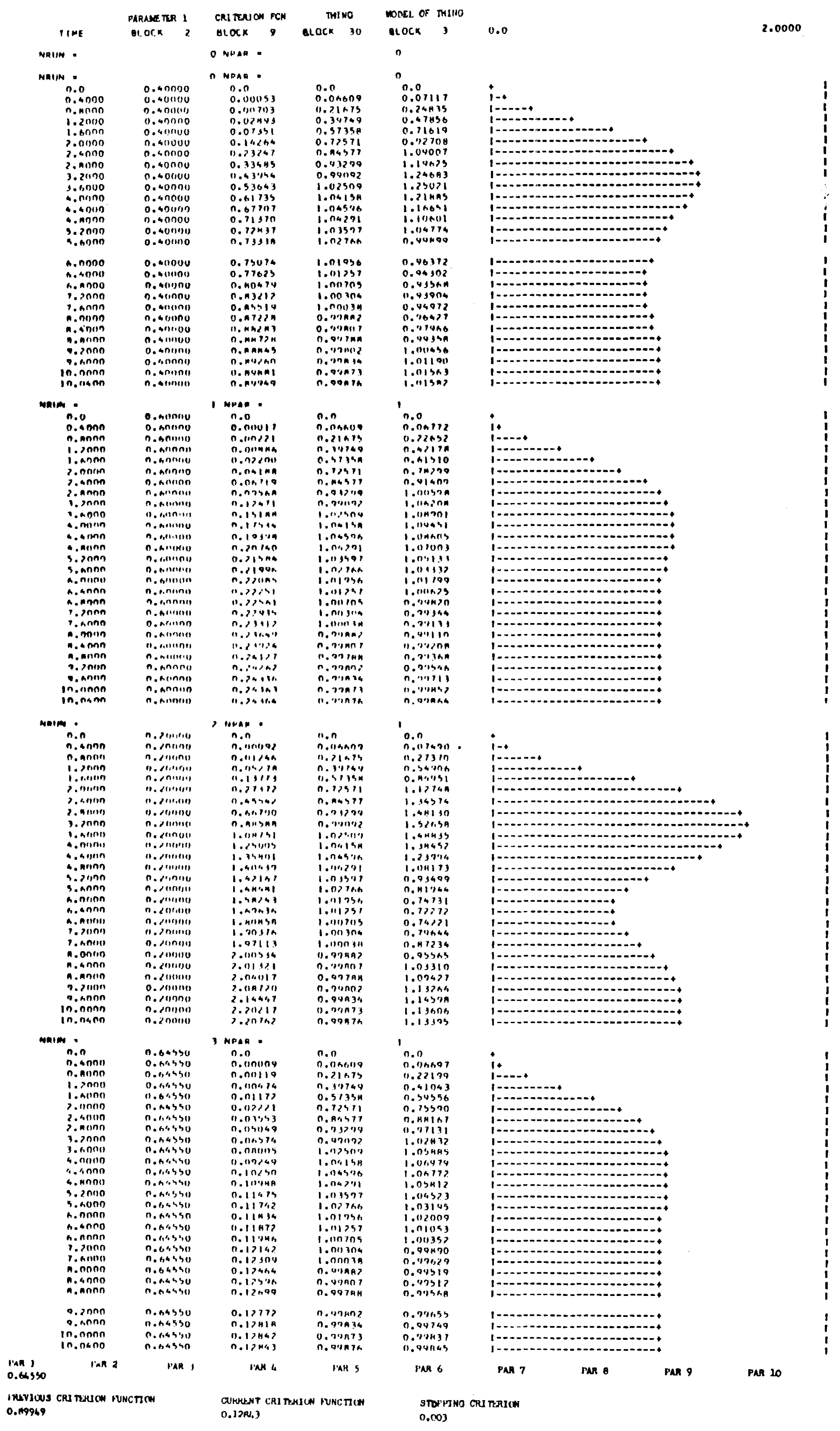
be a model containing an adjustable parameter $\alpha_1$. Further, define an error measure

$$e(t) = y_D(t) - y(t)$$

and a criterion function

$$\phi(\alpha_1) = \int_0^T e^2(t)dt$$

The derivative of the criterion function with respect to the parameter $\alpha_1$ may be computed continuously during the solution.



COMPUTES $\phi(\alpha^0)$

COMPUTES $\phi(\alpha_1+\Delta\alpha_1^+)$

COMPUTES $\phi(\alpha_1-\Delta\alpha_1^-)$ FOLLOWED BY STEEPEST ASCENT PARAMETER STEP $\alpha^{i+1}$

COMPUTES $\phi(\alpha^{i+1})$

SUMMARY OF PREVIOUS 2N+1 SOLUTIONS

```
               PARAMETER 1   CRITERION FCN    THING     MODEL OF THING
      TIME     BLOCK    2    BLOCK    9    BLOCK   30   BLOCK    3     0.0                                  2.0000

NRUN =                       0 NPAR =                   0

NRUN =                       0 NPAR =                   0
      0.0       0.40000       0.0           0.0          0.0
      0.4000    0.40000       0.00053       0.06609      0.07117
      0.8000    0.40000       0.00703       0.21675      0.24835
      1.2000    0.40000       0.02893       0.39749      0.47856
      1.6000    0.40000       0.07351       0.57358      0.71619
      2.0000    0.40000       0.14264       0.72571      0.92708
      2.4000    0.40000       0.23247       0.84577      1.09007
      2.8000    0.40000       0.33485       0.93299      1.19625
      3.2000    0.40000       0.43954       0.99092      1.24683
      3.6000    0.40000       0.53643       1.02509      1.25021
      4.0000    0.40000       0.61735       1.04158      1.21885
      4.4000    0.40000       0.67707       1.04596      1.16651
      4.8000    0.40000       0.71370       1.04291      1.10601
      5.2000    0.40000       0.72837       1.03597      1.04774
      5.6000    0.40000       0.73318       1.02766      0.99899

      6.0000    0.40000       0.75074       1.01956      0.96372
      6.4000    0.40000       0.77625       1.01257      0.94302
      6.8000    0.40000       0.80479       1.00705      0.93568
      7.2000    0.40000       0.83212       1.00304      0.93904
      7.6000    0.40000       0.85519       1.00038      0.94972
      8.0000    0.40000       0.87228       0.99882      0.96427
      8.4000    0.40000       0.88283       0.99807      0.97966
      8.8000    0.40000       0.88728       0.99788      0.99358
      9.2000    0.40000       0.88845       0.99802      1.00456
      9.6000    0.40000       0.89260       0.99834      1.01190
     10.0000    0.40000       0.89881       0.99873      1.01563
     10.0400    0.40000       0.89949       0.99876      1.01582

NRUN =                       1 NPAR =                   1
      0.0       0.60000       0.0           0.0          0.0
      0.4000    0.60000       0.00017       0.06609      0.06772
      0.8000    0.60000       0.00221       0.21675      0.22652
      1.2000    0.60000       0.00886       0.39749      0.42178
      1.6000    0.60000       0.02200       0.57358      0.61510
      2.0000    0.60000       0.04188       0.72571      0.78299
      2.4000    0.60000       0.06719       0.84577      0.91409
      2.8000    0.60000       0.09568       0.93299      1.00578
      3.2000    0.60000       0.12471       0.99092      1.06208
      3.6000    0.60000       0.15188       1.02509      1.08701
      4.0000    0.60000       0.17546       1.04158      1.09451
      4.4000    0.60000       0.19398       1.04596      1.08605
      4.8000    0.60000       0.20740       1.04291      1.07003
      5.2000    0.60000       0.21586       1.03597      1.05133
      5.6000    0.60000       0.21996       1.02766      1.03332
      6.0000    0.60000       0.22085       1.01956      1.01799
      6.4000    0.60000       0.22251       1.01257      1.00625
      6.8000    0.60000       0.22561       1.00705      0.99820
      7.2000    0.60000       0.22935       1.00304      0.99344
      7.6000    0.60000       0.23312       1.00038      0.99133
      8.0000    0.60000       0.23669       0.99882      0.99110
      8.4000    0.60000       0.23926       0.99807      0.99208
      8.8000    0.60000       0.24127       0.99788      0.99368
      9.2000    0.60000       0.24262       0.99802      0.99546
      9.6000    0.60000       0.24336       0.99834      0.99713
     10.0000    0.60000       0.24363       0.99873      0.99852
     10.0400    0.60000       0.24364       0.99876      0.99866

NRUN =                       2 NPAR =                   1
      0.0       0.20000       0.0           0.0          0.0
      0.4000    0.20000       0.00092       0.06609      0.07490 .
      0.8000    0.20000       0.01246       0.21675      0.27370
      1.2000    0.20000       0.05278       0.39749      0.54906
      1.6000    0.20000       0.13773       0.57358      0.84951
      2.0000    0.20000       0.27372       0.72571      1.12748
      2.4000    0.20000       0.45542       0.84577      1.34574
      2.8000    0.20000       0.66790       0.93299      1.48130
      3.2000    0.20000       0.85588       0.99092      1.52658
      3.6000    0.20000       1.08751       1.02509      1.48835
      4.0000    0.20000       1.25005       1.04158      1.38452
      4.4000    0.20000       1.35801       1.04596      1.23994
      4.8000    0.20000       1.40637       1.04291      1.08173
      5.2000    0.20000       1.42167       1.03597      0.93499
      5.6000    0.20000       1.48681       1.02766      0.81944
      6.0000    0.20000       1.58243       1.01956      0.74731
      6.4000    0.20000       1.69636       1.01257      0.72272
      6.8000    0.20000       1.80858       1.00705      0.74221
      7.2000    0.20000       1.90376       1.00304      0.79644
      7.6000    0.20000       1.97113       1.00038      0.87234
      8.0000    0.20000       2.00534       0.99882      0.95565
      8.4000    0.20000       2.01321       0.99807      1.03310
      8.8000    0.20000       2.04017       0.99788      1.09427
      9.2000    0.20000       2.08720       0.99802      1.13264
      9.6000    0.20000       2.14467       0.99834      1.14598
     10.0000    0.20000       2.20217       0.99873      1.13606
     10.0400    0.20000       2.20762       0.99876      1.13395

NRUN =                       3 NPAR =                   1
      0.0       0.64550       0.0           0.0          0.0
      0.4000    0.64550       0.00009       0.06609      0.06697
      0.8000    0.64550       0.00119       0.21675      0.22199
      1.2000    0.64550       0.00474       0.39749      0.41043
      1.6000    0.64550       0.01172       0.57358      0.59556
      2.0000    0.64550       0.02221       0.72571      0.75590
      2.4000    0.64550       0.03553       0.84577      0.88167
      2.8000    0.64550       0.05049       0.93299      0.97131
      3.2000    0.64550       0.06574       0.99092      1.02832
      3.6000    0.64550       0.08005       1.02509      1.05885
      4.0000    0.64550       0.09249       1.04158      1.06979
      4.4000    0.64550       0.10250       1.04596      1.06772
      4.8000    0.64550       0.10988       1.04291      1.05812
      5.2000    0.64550       0.11475       1.03597      1.04523
      5.6000    0.64550       0.11742       1.02766      1.03195
      6.0000    0.64550       0.11834       1.01956      1.02009
      6.4000    0.64550       0.11872       1.01257      1.01053
      6.8000    0.64550       0.11986       1.00705      1.00352
      7.2000    0.64550       0.12142       1.00304      0.99890
      7.6000    0.64550       0.12309       1.00038      0.99629
      8.0000    0.64550       0.12464       0.99882      0.99519
      8.4000    0.64550       0.12596       0.99807      0.99517
      8.8000    0.64550       0.12699       0.99788      0.99568

      9.2000    0.64550       0.12772       0.99802      0.99655
      9.6000    0.64550       0.12818       0.99834      0.99749
     10.0000    0.64550       0.12842       0.99873      0.99837
     10.0400    0.64550       0.12843       0.99876      0.99845
PAR 1        PAR 2        PAR 3        PAR 4        PAR 5        PAR 6        PAR 7        PAR 8        PAR 9        PAR 10
0.64550

PREVIOUS CRITERION FUNCTION          CURRENT CRITERION FUNCTION          STOPPING CRITERION
0.89949                              0.12843                             0.003
```

Figure 4—MOBSSL solutions for first iteration of damping ratio example

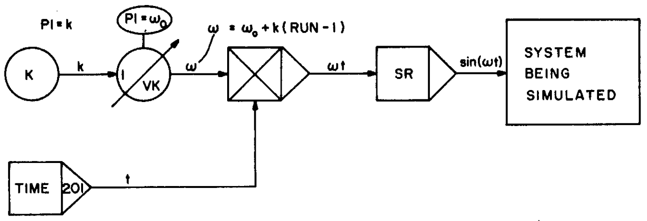


Figure 5—Frequency response implementation through use of the VK element to increment ω

$$\frac{\partial \phi}{\partial \alpha_1} = 2e(t)\,\frac{\partial y(t)}{\partial \alpha_1}$$

where

$$\frac{\partial y(t)}{\partial \alpha_1}$$

is the solution of an associated differential equation described by Meissinger:[7]

$$\frac{d}{dt}\left(\frac{\partial y(t)}{\partial \alpha_1}\right) + \alpha_1\left(\frac{\partial y(t)}{\partial \alpha_1}\right) = -y(t)$$

The parameter adjustment algorithm is

$$\Delta\alpha^i = -K\int_0^T 2e^i(t)\,\frac{\partial y^i}{\partial \alpha_1{}^i}\,dt$$

and

$$\alpha^{i+1} = \alpha^i + \Delta\alpha^1 \qquad i = 1,2,\cdots$$

This process is easily mechanized by the VK element. The incremental changes in the parameter $\alpha_1$ are accumulated from solution to solution. The MOBSSL program is seen in Figure 6. The computational results of this program, for an $\alpha$ of 1.0 are seen in Figure 7. The four columns of tabular data are: time, $y_D(t)$, $y(t)$, $\partial y(t)/\partial\alpha_1$, respectively. The last column and the plot show the growth of the criterion function during the solution. The initial value of the adjustable parameter, $\alpha_1$, is 0.1. It becomes —0.5342, —0.5928, and —0.6365 in three successive solutions, converging towards —1.0. As can be seen from the plot, the error decreases considerably from solution to solution.

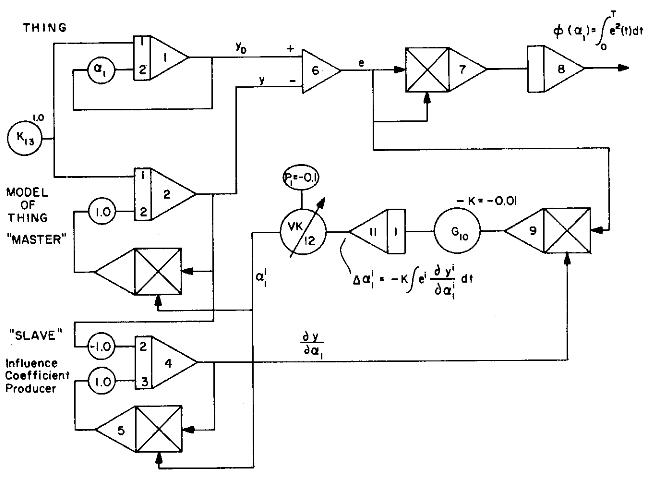

THING
$\phi(\alpha_1) = \int_0^T e^2(t)dt$
α₁
y_D
y
e
K₁₃
1.0
MODEL OF THING "MASTER"
P₁=-0.1
- K = -0.01
VK 12
G₁₀
$\alpha_1^i$
$\Delta\alpha_1^i = -K\int e^i \frac{\partial y^i}{\partial \alpha_1^i} dt$
"SLAVE" Influence Coefficient Producer
-1.0
$\frac{\partial y}{\partial \alpha_1}$

Figure 6—MOBSSL block diagram demonstrating use of VK block as iterative element in parameter search by discrete sensitivity difference equation technic

## The STORAGE element

The STORAGE element is very similar to the VK element, except that its output remains constant for one or more solutions depending on the block's parameters. During the first solution, the output of the STORAGE blocks ares et equal to their first parameter's values. Thereafter they are set equal to the sum of their inputs. The change in output takes place every P3 solutions, when

$$[(N-P2)\text{ MODULO P3}] = 0$$

For example, if P2 = 0 and P3 = 3, then the output of the storage block will be P1 during solutions one, two and three and will be reset to the sum of its inputs between solutions three and four. It will hold this output during solutions four, five and six, recomputing its output between solutions six and seven, etc. Notice that the STORAGE element does not accumulate in computing its new output. If P2 = 0 and P3 = 1 the STORAGE element is a non-accumulating VK element. In Figure 8, the S element is used to pass the output of integration element 1 from one solution to the next. During the first solution the output of the S element is 3.0. A more complex application is shown in Figure 9 where the output of the S element is used to modify the structure of the simulation from solution to solution. In odd numbered solutions, N = 1,3,5, etc., the output of the S element will be +1 and the output of RELAY element 4 will be —1. As with all

```
MOBSSL,UAF-- MERRITT'S OWN BLOCK STRUCTURED SIMULATION LANGUAGE, UNPRONOUNCEABLE ACRONYM FOR...MK II MOD 2   JAN 01 1969

CONFIGURATION SPECIFICATIONS

OUTPUT NAME       BLOCK NUMBER   BLOCK TYPE   INPUT 1   INPUT 2   INPUT 3
THING                  1             I           13         1        0
MODEL OF THING         2             I           13         3        0
                       3             X            2        12        0
INFLUENCE COEFF.       4             I            0         2        5
                       5             X           12         4        0
ERROR                  6             +            1        -2        0
ERROR SQUARED          7             X            6         6        0
CRITERION FUNCTN       8             I            7         0        0
                       9             X            6         4        0
GAIN                  10             G            9         0        0
DELTA ALPHA           11             I           10         0        0
VARIABLE CONSTNT      12             VK          11         0        0
INPUT                 13             F            0         0        0

                        INITIAL CONDITIONS AND PARAMETERS

IC/PAR NAME     BLOCK      IC/PAR1         PAR2          PAR3
ALPHA              1          0.0       -1.00000    0.0
                   2          0.0        1.00000    0.0
                   4          0.0       -1.00000    1.00000
                  10       -0.02500    0.0          0.0
                  12       -0.10000    0.0          0.0
                  13        1.00000    0.0          0.0

                        PROGRAM MODE       RUN
INTEGRATION INTERVAL IS               0.01000
TOTAL TIME IS                 4.20000
PRINT INTERVAL IS                0.30000
BLOCKS TO BE PRINTED ARE          1          2


BLOCK TO BE PLOTTED  IS          8 RANGE OF PLOTTED VARIABLE IS          0.0          10.00000




                THING       MODEL OF THING  INFLUENCE COEFF  CRITERION FCN
   TIME        BLOCK   1     BLOCK   2       BLOCK   4       BLOCK   8      0.0                                      10.000
   0.0          0.0          0.0             0.0             0.0            +                                              I
   0.3000       0.25918      0.29554        -0.04411         0.00008        +                                              I
   0.6000       0.45119      0.58235        -0.17296         0.00221        +                                              I
   0.9000       0.59343      0.86068        -0.38150         0.01425        +                                              I
   1.2000       0.69880      1.13078        -0.66490         0.05119        +                                              I
   1.5000       0.77687      1.39287        -1.01856         0.13400        +                                              I
   1.8000       0.83470      1.64722        -1.43805         0.28767        I+                                             I
   2.1000       0.87754      1.89406        -1.91919         0.53936        I-+                                            I
   2.4000       0.90928      2.13359        -2.45797         0.91691        I---+                                          I
   2.7000       0.93279      2.36605        -3.05057         1.44768        I-------+                                      I
   3.0000       0.95021      2.59164        -3.69333         2.15788        I----------+                                   I
   3.3000       0.96311      2.81056        -4.38276         3.07207        I--------------+                               I
   3.6000       0.97267      3.02301        -5.11556         4.21287        I--------------------+                         I
   3.9000       0.97975      3.22918        -5.88855         5.60087        I----------------------------+                 I
   4.2000       0.98500      3.42926        -6.69872         7.25461        I-------------------------------------+        I
   4.2100       0.98515      3.43583        -6.72633         7.31452        I-------------------------------------+        I

THE VALUE OF VARIABLE CONSTANT WITH BLOCK NO.  12 IS               -0.5342 AT END OF RUN   1
RUN TERMINATED AT TIME EQUAL TO TOTAL TIME
   0.0          0.0          0.0             0.0             0.0            +                                              I
   0.3000       0.25918      0.27719        -0.04047         0.00002        +                                              I
   0.6000       0.45119      0.51334        -0.14579         0.00051        +                                              I
   0.9000       0.59343      0.71452        -0.29587         0.00307        +                                              I
   1.2000       0.69880      0.88591        -0.47514         0.01027        +                                              I
   1.5000       0.77687      1.03192        -0.67167         0.02505        +                                              I
   1.8000       0.83470      1.15628        -0.87641         0.05014        +                                              I
   2.1000       0.87754      1.26224        -1.08261         0.08772        +                                              I
   2.4000       0.90928      1.35250        -1.28534         0.13932        +                                              I
   2.7000       0.93279      1.42939        -1.48111         0.20576        I+                                             I
   3.0000       0.95021      1.49490        -1.66756         0.28727        I+                                             I
   3.3000       0.96311      1.55071        -1.84313         0.38362        I-+                                            I
   3.6000       0.97267      1.59826        -2.00698         0.49418        I-+                                            I
   3.9000       0.97975      1.63876        -2.15871         0.61811        I--+                                           I
   4.2000       0.98500      1.67327        -2.29832         0.75440        I---+                                          I
   4.2100       0.98515      1.67432        -2.30277         0.75915        I---+                                          I

THE VALUE OF VARIABLE CONSTANT WITH BLOCK NO.  12 IS               -0.5928 AT END OF RUN   2
RUN TERMINATED AT TIME EQUAL TO TOTAL TIME
   0.0          0.0          0.0             0.0             0.0            +                                              I
   0.3000       0.25918      0.27484        -0.04000         0.00002        +                                              I
   0.6000       0.45119      0.50490        -0.14251         0.00038        +                                              I
   0.9000       0.59343      0.69748        -0.28609         0.00228        +                                              I
   1.2000       0.69880      0.85869        -0.45464         0.00757        +                                              I
   1.5000       0.77687      0.99363        -0.63621         0.01829        +                                              I
   1.8000       0.83470      1.10657        -0.82209         0.03631        +                                              I
   2.1000       0.87754      1.20111        -1.00604         0.06303        +                                              I
   2.4000       0.90928      1.28024        -1.18375         0.09934        +                                              I
   2.7000       0.93279      1.34648        -1.35237         0.14565        +                                              I
   3.0000       0.95021      1.40193        -1.51015         0.20196        I+                                             I
   3.3000       0.96311      1.44835        -1.65617         0.26793        I+                                             I
   3.6000       0.97267      1.48721        -1.79004         0.34302        I+                                             I
   3.9000       0.97975      1.51973        -1.91187         0.42653        I-+                                            I
   4.2000       0.98500      1.54696        -2.02202         0.51770        I-+                                            I
   4.2100       0.98515      1.54779        -2.02549         0.52086        I-+                                            I

THE VALUE OF VARIABLE CONSTANT WITH BLOCK NO.  12 IS               -0.6365 AT END OF RUN   3
RUN TERMINATED AT TIME EQUAL TO TOTAL TIME
   0.0          0.0          0.0             0.0             0.0            +                                              I
   0.3000       0.25918      0.27309        -0.03966         0.00001        +                                              I
   0.6000       0.45119      0.49872        -0.14011         0.00030        +                                              I
   0.9000       0.59343      0.68512        -0.27903         0.00178        +                                              I
```

Figure 7—MOBSSL data and computational results from first three iterations of parameter search by sensitivity equation method

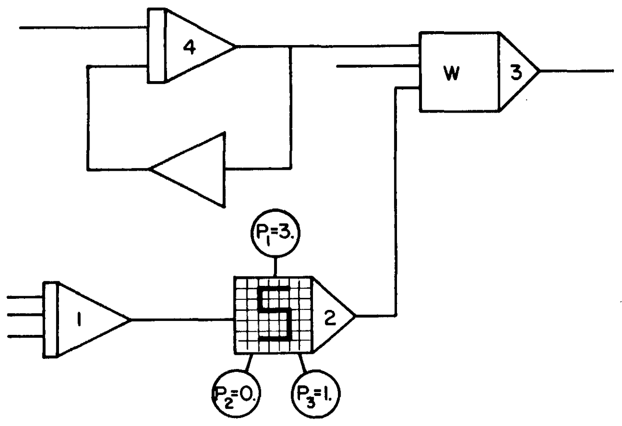


Figure 8—Use of storage element to store and transfer integrator output between simulations during iterative operation

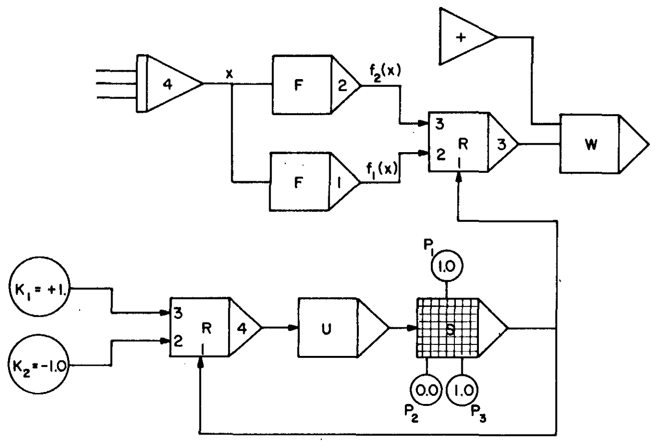


Figure 9—Storage element used to modify structure of simulation from one solution to next

block structured languages, the sorting algorithms experience difficulty with purely algebraic loops. The STORAGE-RELAY element loop is rendered sortable by the inclusion of the UNIT DELAY element. The UNIT DELAY has no effect on the computations, and the output of RELAY element 3 will be $f_1(x)$. During even numbered solutions the output of the STORAGE element will be $-1$ and the output of RELAY element 3 will be $f_2(x)$.

*Hybrid computational elements*

MOBSSL, UAF has been developed for use in the USC System Simulation Laboratory. The System Simulation Laboratory's computer complement is shown in Figure 10. The software and hardware link-

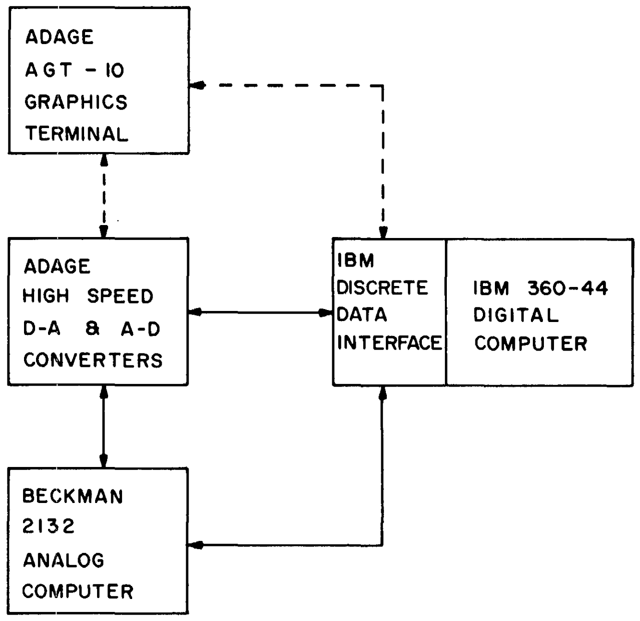

ADAGE AGT - 10 GRAPHICS TERMINAL
ADAGE HIGH SPEED D-A & A-D CONVERTERS
IBM DISCRETE DATA INTERFACE
IBM 360-44 DIGITAL COMPUTER
BECKMAN 2132 ANALOG COMPUTER

Figure 10—USC system simulation laboratory computer complement

age between the computer graphics terminal and the IBM 360 is not yet installed. The analog computer is equipped with a multi-channel strip chart recorder, one and two pen x-y recorders as well as oscilloscopes and memoscopes. Software presently exists to allow the digital computer to carry out the following interface operations:

a. digital to analog conversions
b. analog to digital conversions
c. read discrete data lines
d. set discrete output lines
e. control the mode of the analog computer
f. operate the analog computer's select system
g. process external interrupt signals
h. set potentiometers in the analog computer

As yet, only the first two functions analog to digital and digital to analog conversions, with element designations of AD and DA respectively, are available within MOBSSL. MOBSSL programs may contain up to 10 DA elements and up to 32 AD elements, limited only by the available hardware The hybrid elements may be used separately as I/O elements or together as part of a closed loop hybrid operation.

The DA element is often used as an output element in MOBSSL simulations. As shown in Table I, the DA element causes a voltage, equal to its first input, to appear at the output of the digital to analog converter selected by its first parameter. If the input exceeds

±100.0, the output voltage will not be correct. DA elements may be used to drive recording devices in order to obtain graphical presentations of MOBSSL results. Because of large variations in computation times and input-output times, unless special timing routines are used, the amount of real time between successive outputs will not be constant during a solution. There are several ways of getting around this.

1. Use the graphical results qualitatively and obtain quantitative results from the printer listing.
2. If a multichannel strip chart recorder is used, place a known function of time on one channel and derive timing information from it. The independent variable, sine waves, output of timing elements, etc., are convenient signals.
3. If an x-y plotter is used place the independent variable, the output of block 201, on one axis.
4. Two dependent variables are being plotted against each other and no timing information is required.

Methods 3 and 4 are used in the example described below.

The AD element type is useful for changing parameters and initial conditions. As shown in Table I, the input to an AD block is supplied by an ADC located on the analog computer patchboard. Parameter 1 of the AD block determines the ADC number. The AD block output is a floating point number between ±100.0. If the input exceeds ±100.0 volts, the output of the AD block will be incorrect. On line parameter changes can be achieved by connecting the outputs of manually operated potentiometers to the input of an A-D converter as shown in Figure 11a. Figure 11b demonstrates the use of the AD block to permit on-line adjustment of constants and coefficients appearing in MOBSSL block diagrams. Figure 11c demonstrates the use of the AD block to allow on-line changes in integrator initial conditions. This is valid since the output of an integrator is:

$$e_0(t) = e_0(0) + \int_0^t e_{in}(t)\,dt$$

and $e_0(0)$ can be any number summed with the output of an integrator having zero as its "initial condition."

When MOBSSL is being used in an iterative mode, on-line adjustments are needed only at the beginning of a solution. Parameter variations during the solution are undesirable. This may be achieved by using the

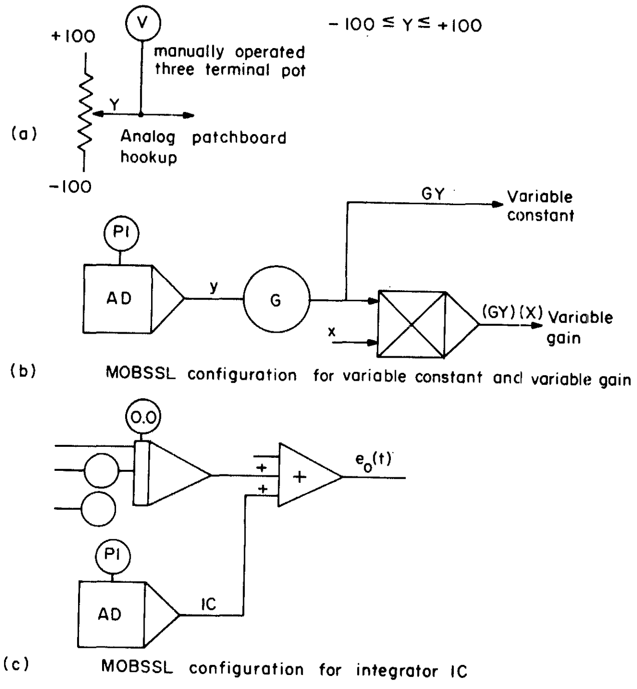

+100
V
-100 ≤ Y ≤ +100
manually operated three terminal pot
Y
(a)
Analog patchboard hookup
-100
GY
Variable constant
PI
AD
y
G
(GY)(X) Variable gain
x
(b)
MOBSSL configuration for variable constant and variable gain
0.0
+
+
+
$e_0(t)$
PI
AD
IC
(c)
MOBSSL configuration for integrator IC

Figure 11—Use of AD element for on-line parameter changes

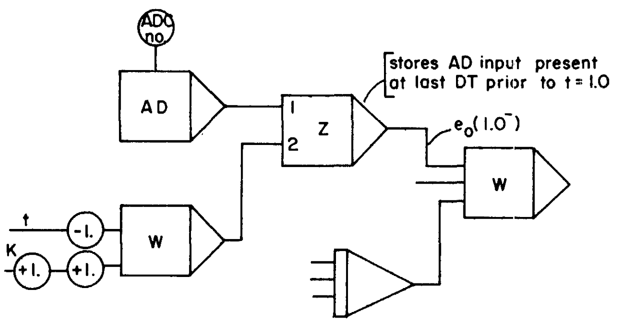


Figure 12—AD block used to modify a parameter at the beginning of a run

zero order hold as a sample and store element. When input number 2 to the ZOH element is less than or equal to zero it holds its previous output. When it is positive it samples, stores and holds present input. In the example shown in Figure 12, the ADC is effective only during the first second of the solution, after which it may be pre-set in preparation for the next solution.

An AD element can be used as the input to a QUIT block to terminate a run from the analog console.

Other applications of the AD element include sampling and processing of analog data where synchronous sampling is not required. The output of the gaussian noise generator, both direct and filtered, located in the Beckman Analog Computer, may be sampled and used in place of the output of the uniform distribution Random Number Generator block type.

Attempts to use the hybrid block elements in real time applications have brought to light the need for a whole series of timing and interrupt processing elements. These elements will expand the real time capability of MOBSSL considerably.

The following example often referred to as the Host-Parasite problem, demonstrates the use of the DA block to drive an X-Y plotter. It is a set of differential equations which represents the population of hosts and parasites as a function of time. The physical situation from which the differential equations are abstracted comes about when there is a host (i.e., food for a parasite) which would reproduce at a known rate if there were no parasites. The parasites die off at a known rate if there are no hosts. Finally, a decrease in the number of hosts and an increase in the number of parasites is a function of the number "encounters" between hosts and parasites. Whenever a host is unlucky enough to encounter a parasite, the parasite eats him up. The equations implemented are:

$$\dot{H} = K_1H - K_4HP$$

$$\dot{P} = K_2H + K_3HP$$

| | |
|---|---|
| $H \triangleq$ | host population as a function of time |
| $P \triangleq$ | parasite population as a function of time |
| $K_1 \triangleq$ | overall growth rate of hosts per hour assumirg no parasites |
| $K_2 \triangleq$ | overall decay rate of parasites per hour assuming no hosts |
| $K_3, K_4 \triangleq$ | number of host-parasite encounters per hour |
| $t \triangleq$ | time in hours |

where

$K_1 \triangleq 0.05/\text{hour}$ (+5% per hour)

$K_2 \triangleq 0.10/\text{hour}$ (−10% per hour)

$K_3 \triangleq 2 \times 10^{-4}/\text{host-hour}$ ⎫ oneen counter per 5000 hours

$K_4 \triangleq 2 \times 10^{-4}/\text{parasite-hour}$ ⎭ for every host-parasite pair

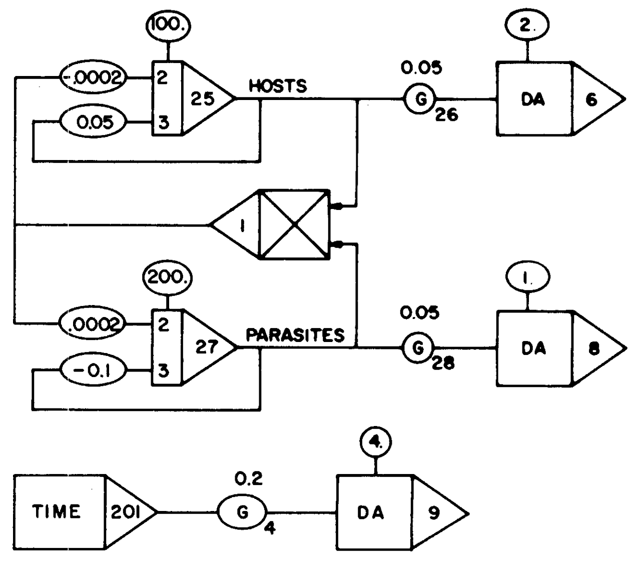


Figure 13—MOBSSL block diagram for the host-parasite problem

Initial Conditions:

| | Run I | Run II | Run III | Run IV |
|---|---|---|---|---|
| H(0) | 100 | 1200 | 600 | 500 |
| P(0) | 200 | 1200 | 500 | 250 |

The MOBSSL diagram is shown in Figure 13, a listing of the MOBSSL configuration specifications, parameters and other simulation data are shown in Figure 14. Figure 15 is a graph of hosts vs. time and parasites vs. time obtained using the PCHG mode and interchanging parameter 1 of blocks 6 and 8 on the second run. Time is obtained from DA block 9 appropriately scaled by gain block 4 from block 201 which provides the independent variable. Figure 16 is a phase plane plot of hosts vs. parasites for four sets of IC's. DA 6 provides the input for the plotter's X axis and DA 8 drives the plotter's Y axis. Note that the existence of closed orbits for all physically realizable IC's is clearly demonstrated, as well as the existence of a stationary point at (H,P) = (500,250).

### *Disk input and disk output elements*

Through the use of the Disk Input, DI and Disk Output, DO, blocks vector functions of the independent variable may be respectively read out of and written into previously alloc·ted data sets on disk storage during a simulation. The DI block is used when

```
MOBSSL,UAF-- MERRITT'S OWN BLOCK STRUCTURED SIMULATION LANGUAGE, UNPRUNOUNCEABLE ACRONYM FOR...MK II MOD 2   JAN 01 1969

CONFIGURATION SPECIFICATIONS

OUTPUT NAME       BLOCK NUMBER   BLOCK TYPE    INPUT 1    INPUT 2    INPUT 3
MULTIPLIER              1             X           25         27         0
HOSTS                  25             I            0          1        25
PARASITES              27             I            0          1        27
HOST   SCALING         26             G           25          0         0
PARASITE SCALING       28             G           27          0         0
HOST  DAC               6            DA           26          0         0
PARASITE DAC            8            DA           28          0         0
TIME  SCALING           4             G          301          0         0
TIME  DAC               9            DA            4          0         0

                         INITIAL CONDITIONS AND PARAMETERS

IC/PAR NAME       BLOCK      IC/PAR1           PAR2           PAR3
PARASITE IC         27       200.00000    0.00020   -0.10000
HOST  IC            25       100.00000   -0.00020    0.05000
HOST  SCALING       26         0.05000    0.0        0.0
PARASITE SCALING    28         0.05000    0.0        0.0
HOST DAC NUMBER      6         2.00000    0.0        0.0
PARASITE DAC NO.     8         1.00000    0.0        0.0
TIME DAC NUMBER      9         4.00000    0.0        0.0
TIME SCALING         4         0.20000    0.0        0.0

                         PROGRAM MODE      STOP
INTEGRATION INTERVAL IS                0.10000
TOTAL TIME IS               318.00000
PRINT INTERVAL IS                 6.00000
BLOCKS TO BE PRINTED ARE          4           1           27


BLOCK TO BE PLOTTED  IS          25 RANGE OF PLOTTED VARIABLE IS                 0.0               1640.00000


                T SCALING    MULTIPLIER      PARASITES      HOSTS
   TIME         BLOCK   4     BLOCK   1      BLOCK  27      BLOCK  25       0.0                                              1640.0
   0.0          0.0        20000.00000     200.00000      100.00000       I--+                                                I
   6.0000       1.20000    13882.89062     124.48114      111.52608       I--+                                                I
  12.0000       2.40000    10557.38677      79.05423      133.54616       I---+                                               I
  18.0000       3.60000     8659.57031      51.89128      166.87917       I----+                                              I
  24.0000       4.80000     7643.46094      35.73289      213.90549       I-----+                                             I
  30.0000       6.00000     7319.06250      26.29605      278.33325       I-------+                                           I
  36.0000       7.20000     7735.94922      21.17857      365.27246       I----------+                                        I
  42.0000       8.40000     9267.69922      19.25253      481.37573       I-------------+                                     I
  48.0000       9.59999    13045.33594      20.55446      634.67187       I------------------+                                I
  54.0000      10.79999    22565.01562      27.08324      833.17285       I------------------------+                          I
  60.0000      11.99999    50248.58203      46.60127     1078.26636       I-------------------------------+                   I
  66.0000      13.19999   145855.68750     109.20946     1335.55957       I---------------------------------------+           I
  72.0000      14.39999   463270.18750     325.18359     1424.64209       I------------------------------------------+        I
  78.0000      15.59999   796153.37500     804.10059      990.11670       I-----------------------------+                     I
  84.0000      16.79999   426875.31250    1001.42114      426.26953       I------------+                                      I
  90.0000      17.99998   150926.87500     777.02710      194.23636       I-----+                                             I
  96.0000      19.19998    67226.98828     511.20435      121.72626       I--+                                                I
 102.0000      20.39998    32157.91016     319.64697      100.60446       I--+                                                I
 108.0000      21.59998    19788.89062     197.59003      100.15128       I--+                                                I
 114.0000      22.79999    13772.43750     123.07399      111.94923       I--+                                                I
 120.0000      23.99998    10494.70703      78.18076      134.23650       I---+                                               I
 126.0000      25.19998     8623.89453      51.36963      167.87926       I----+                                              I
 131.9999      26.39998     7626.52734      35.42406      215.29242       I-----+                                             I
 137.9999      27.59998     7319.18359      26.11966      280.21753       I-------+                                           I
 143.9999      28.79997     7757.67578      21.09200      367.80200       I----------+                                        I
 149.9999      29.99998     9326.94141      19.24123      484.73755       I--------------+                                    I
 155.9999      31.19998    13189.30859      20.63794      639.08081       I------------------+                                I
 161.9999      32.39998    22947.12891      27.35725      838.79517       I------------------------+                          I
 167.9999      33.59998    51449.55469      47.42339     1084.89844       I-------------------------------+                   I
 173.9999      34.79997   150217.43750     112.00195     1341.20386       I---------------------------------------+           I
 179.9999      35.99997   475206.37500     334.35937     1421.24438       I------------------------------------------+        I
 185.9999      37.19998   794302.37500     815.88574      973.54614       I----------------------------+                      I
 191.9999      38.39998   416341.50000     999.07373      416.72754       I------------+                                      I
 197.9999      39.59998   147190.50000     769.86987      191.18887       I-----+                                             I
 203.9999      40.79997    61041.15234     505.40210      120.77740       I--+                                                I
 209.9999      41.99997    31706.80859     315.81982      100.39525       I--+                                                I
 215.9999      43.19997    19582.57031     195.22153      100.30949       I--+                                                I
 221.9999      44.39998    13664.20312     121.59250      112.37703       I--+                                                I
 227.9999      45.59998    10443.24609      77.32295      134.93082       I---+                                               I
 233.9999      46.79997     8588.98437      50.85753      168.88330       I----+                                              I
 239.9999      47.99997     7610.15625      35.12105      216.68373       I-----+                                             I
 245.9999      49.19997     7319.84375      25.94704      282.10718       I-------+                                           I
 251.9999      50.39998     7780.01562      21.00793      370.33716       I----------+                                        I
 257.9998      51.59995     9387.41406      19.23728      488.10718       I--------------+                                    I
 263.9998      52.79994    13336.05469      20.72440      643.49561       I------------------+                                I
 269.9998      53.99994    23337.57812      27.63748      844.41772       I------------------------+                          I
 275.9998      55.19994    52681.76172      48.26521     1091.50586       I-------------------------------+                   I
 281.9998      56.39993   154698.06250     114.86980     1346.72583       I---------------------------------------+           I
 287.9998      57.59995   487194.18750     343.70508     1417.47754       I------------------------------------------+        I
 293.9998      58.79994   791793.25000     827.36304      957.00830       I----------------------------+                      I
 299.9998      59.99994   406039.06250     996.45972      407.48169       I------------+                                      I
 305.9998      61.19994   143584.06250     762.74780      188.24585       I-----+                                             I
 311.9998      62.39993    59892.84766     499.68262      119.86179       I--+                                                I
 317.9998      63.59995    31267.78516     312.05664      100.19908       I--+                                                I
 318.0999      63.61996    30981.00000     309.57080      100.07727       I--+                                                I
RUN TERMINATED AT TIME EQUAL TO TOTAL TIME
  STOP      0
/&       END OF JOB
```

Figure 14—MOBSSL printer listing for the host-parasite problem

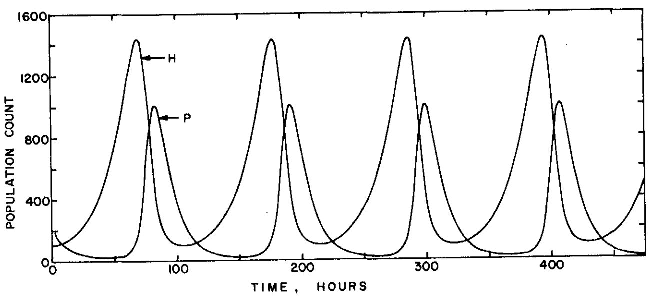


Figure 15—X-Y plotter graph of hosts and parasites vs time

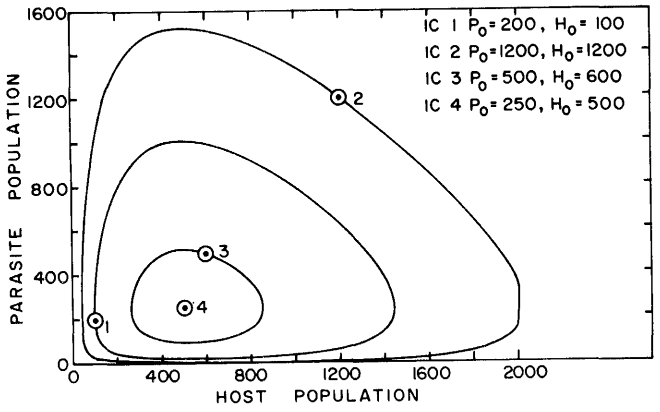


Figure 16—Phase plane plot of hosts vs parasites for four sets of initial conditions

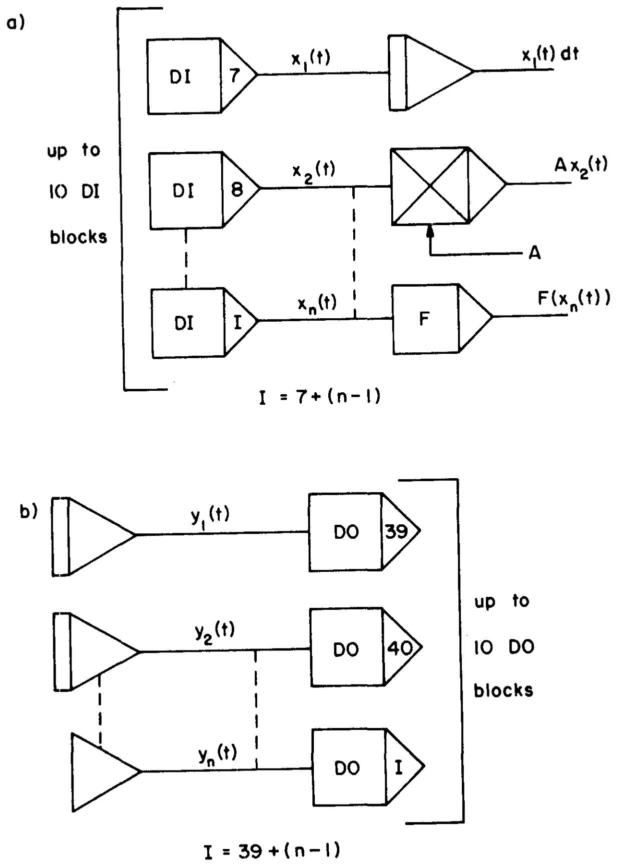


Figure 17—Disk input and disk output block configurations

a data set stored on a *disk* serves as an *input* to the MOBSSL simulation. These data could be stored at any prior time including the immediately preceding simulation during the present job if MOBSSL is in an interative simulation mode. The DO block is used when it is desired to write block *outputs* into a data set stored on a *disk*. The user therefore has the necessary tools to

1. Provide complex previously obtained vector valued inputs to a MOBSSL simulation.
2. Store vector valued MOBSSL time histories for future use.
3. Perform various functional optimization techniques such as quasilinearization in which the previous time history serves as data for the present solution.

Up to 10 DI blocks connected as shown in Figure 17a are permitted. DI block numbers must be sequential and ordered to correspond to stored disk data sequence, i.e., the lowest block number corresponds to the first block output stored in a record, next to lowest block number to the second block output stored in a record, etc.

Similarly, up to 10 DO blocks connected as shown in Figure 17b are permitted. DO block numbers must be sequential and ordered to correspond to desired disk data sequence i.e., the output of the DO with the lowest block number corresponds to the first variable stored in each disk data set record, etc.

Disk read and write time intervals are independently determined by two user supplied entries on a Sample Time card which also includes line printer time interval. DI block outputs remain the same until the number of MOBSSL complete integration cycles since the most recent disk read multiplied by the integration inerval equals the disk read time. Similarly, writing onto the disk occurs only when the accumulated MOBSSL time interval since the most recent disk write is equal to the disk write time.

### *Graphic MOBSSL*

The man-simulation language interface can be improved considerably through the application of computer graphics terminals. These devices allow block diagrams and equations to be manipulated on the screen of a cathode ray tube using light pens, tablets and alpha-numeric keyboards. When the problem definition is completed, the simulation language is called to generate the desired solutions. As the solutions are computed, they are displayed by the computer graphics terminal.

Two graphics programs are presently under development at USC. The first, a graphic block diagram editing program, allows the user to construct MOBSSL diagrams on the screen of the graphics terminal. The second, a differential equation editing program, allows systems of equations to be drawn on the screen. Prior to execution of MOBSSL, the equations are translated into a MOBSSL block diagram. The user may view the resultant diagram or immediately enter the MOBSSL program. Subsequent editing operations may be carried out on either the equations or the corresponding block diagram.

These two problem preparation programs provide users with an extremely flexible communication interface. The speed with which large amounts of instructional and reference material may be displayed makes it possible to operate these programs with almost no prior instruction.

### *Future plans for MOBSSL*

Future developments of MOBSSL will be directed toward improved man-computer communications. These will take the form of additional process oriented block elements; additional parameter and functional optimization and identification procedures, timing and interrupt processing elements, and expanded graphics facilities.

## ACKNOWLEDGMENT

The research described in this paper was carried out under NSF Grant No. GK 2716 and NASA Grant No. NGR-05-018-022. The authors wish to acknowledge the assistance provided to them by the Staff of the System Simulation Laboratory: M. Asa G. Hammerwold, A. Lew, C. Hartman and J. Maloney.

## REFERENCES

1 J J CLANCY M S FINEBERG
*Digital simulation languages: A critique and a guide*
Proc SJCC Vol 27 Part 1 23-36

2 R D BRENNAN H SANO
*PACTOLUS*
Proc FJCC 1964 Vol 26 Spartan Books Inc

3 *1130 continuous system modeling program (1130-CX-13X) program reference manual*
H20-02820 IBM Corp

4 R T HARNETT F J SANSOM
*MIDAS programming guide*
Rpt No SEG-TDR-64-1 Wright Patterson AFB Ohio 1964

5 *System 360 continuous system modeling program (360A-CX-16X) User's manual*
H20-0367-2 IBM Corp

6 G A BEKEY W J KARPLUS
*Hybrid computation*
John Wiley and Sons 1968

7 H F MEISSINGER
*The use of parameter influence coefficients in computer analysis of dynamic systems*
Proc Western Joint Computer Conf Vol 17 1960 181-192

# A hybird computer programming system

*by* M. A. FRANKLIN and J. C. STRAUSS

*Carnegie-Mellon University*
Pittsburgh, Pennsylvania

## INTRODUCTION

In order to analyze and subsequently synthesize complex systems, engineers have increasingly turned to computer simulation techniques. Until recently, simulation techniques could generally be divided either on the basis of the type of computer to be used, or the type of system to be simulated.

Of the two types of computers available, the analog computer was usually restricted to simulation of continuous systems (i.e., systems described by sets of differential equations), while the digital computer was used primarily in simulation of discrete event probabilistic systems. As numerical integration techniques improved and digital computer speed increased, however, it became clear that it was possible to solve sets of differential equations and thus simulate continuous systems on the digital computer. Many digital computer programming systems have been designed for use in such simulations.

Associated with digital computer simulation there thus arose two types of simulation languages; one for describing discrete event probabilistic systems,[1,2] and one for describing continuous systems.[3,4] The main concern of this paper is with continuous systems simulation.

With two types of computers available for simulating continuous systems the question of which type is "best" presents itself. The answer is often unclear and is certainly dependent on the system to be simulated and the definition of the word "best".[5] In general the digital computer has the advantages of high accuracy combined with low problem setup costs and the availability of extensive digital computer logic and automatic programming facilities. It has the disadvantages of limited problem solution speed (dynamic response) and limited man/machine interaction capability. The analog computer, on the other hand, has the advantages of high dynamic response and good man/machine interaction with the disadvantages of limited accuracy and negligible automatic problem setup and programming facilities.

More recently with the advent of the hybrid computer, an attempt has been made to combine in one computer system the advantages of both the analog and digital computers. Unfortunately, in addition to the advantages, some of the disadvantages have been combined and some entirely new problems have arisen. At this point, however, it appears that with the added facilities available in a hybrid computer system these problems can be overcome, and an overall improvement in continuous system simulation capabilities achieved.

One of the main problems encountered with hybrid computers is that of effective system utilization. The process of effecting hybrid simulation studies can be described in the four phases presented in Table I. It has been found, from experience, that often the man hours spent in off line problem preparation and on line problem setup and debugging, exceed the man hours spent in determining solution methods and actually performing the simulation. In addition, it is often the case that more computer time is spent during the setup and debugging process than is spent during problem execution. These disadvantages derive mainly from the analog computer part of the programming and have, in general, carried over to the hybrid en-

TABLE I—Phases in preparing a problem for hybrid computer solution

1. Defining the Problem and Determining the Solution Method
   a. Equations which govern the system
   b. Parameters to be varied in the simulation study
   c. General solution procedure
2. Off Line Problem Preparation
   a. Allocating the problem to analog or digital parts of hybrid
   b. Scaling the equations
   c. Determining the static check values
   d. Allocating components to the analog patchboard
   e. Allocating interface linkages
   f. Developing the necessary digital programs
3. On Line Preparation and Debugging of Problem
   a. Wiring the patchboard
   b. Loading the digital computer programs
   c. Static checking of analog and interface components
   d. Dynamic checking of analog and interface components
4. Performing the Simulation Study
   a. Run time interaction; changing parameters, termination problem, etc.
   b. Run time diagnostics
   c. Run time documentation

vironment. In part the reason for this is that, unlike the all digital computer system, the hybrid computer system lacks a coherent overall programming framework in which to operate.

Several programming systems have been developed in the past to help automate the tasks of problem preparation, setup and debugging. The most prominent among these are the APACHE[7,8,9] and HYTRAN[10] systems. Both of these systems have had their drawbacks. The APACHE system has had limited operational success in this country[11,12] due to a combination of poor documentation and a lack of clearly defined program modularity. More fundamental criticisms of APACHE relate to its inability to handle either the extensive parallel logic capabilities available on many current analog computers or the variety of interface elements available on hybrid computers. In addition, the lack of a macro-programming capability, the difficulties in extending APACHE for use on machines other than the IBM 7090 and PACE 231-R, and the general lack of a suitable overall hybrid programming framework appear to make modification of the APACHE system for current hybrid computers an uneconomical, if not impossible, venture. The HYTRAN system, which performs a subset of those tasks handled by APACHE, has also had limited success. In this case the reason appears to be that, for most users, the time required to prepare the large quantity of input information demanded by the system did not justify the system's return.

In this paper a hybrid programming system is proposed which attempts to avoid the drawbacks of the APACHE and HYTRAN systems while creating an overall programming framework in which to view hybrid computer operations. The system is presented in terms of four distinct language levels and the processing between these levels. The language levels range from a highest level machine independent "source language" for general problem representation, to a "machine language" which represents the actual hybrid computer implementation. The processing between the language levels represents clearly defined tasks which can be modularly implemented in successive stages. At present, the two lowest levels, the "assembly language" and the "machine language" have been completely specified and the processing between them has been implemented for an EAI 680[6] Analog/Logic Computer. Some of the specification details are presented and a programming example involving the simulation of an automobile suspension system is provided.

## *Characteristics and concepts of hybrid programming systems*

This section considers characteristics which an effective hybrid programming system should possess and some concepts which aid in effecting such a system.

1. *Language Levels*: One of the keys to the successful development of effective all digital computer systems has been the familiar concept of language levels. Typically the lowest level, a machine language, is present for describing the most basic operations which can be performed on the computer in terms of the actual performing computer hardware. The next higher level language, an assembly language, deals with these basic operations, using mnemonics and symbolic notation. At a still higher level a source language consists of symbolic operations which represent combinations of basic operations. In the few existing analog oriented systems this approach has been avoided and there has been an emphasis on higher level language development. This has been done without developing clear machine and assembly languages for the representation of analog operations.

Part of the reason for this is that machine language representations of digital problems have direct operational meaning in terms of sequencing and logical decoding of digital instructions, while such representations for an analog computer, at least at present, have only symbolic significance. Symbolic representational ability is, however, important both from a conceptual and practical point of view. In particular one of the principal criticisms of APACHE was its inability to respond to the analog programmers ability to think up new and sometimes strange circuit patching.

A language level approach to a hybrid programming system which provides the ability on the lowest level to represent all legal analog computer patching thus seems desirable.

2. *Language Level Mixing*: Provision should be made for intermixing language levels. While writing in the source language it should be possible to mix in assembly or machine language statements, thus enabling the specification of particular circuit patching for special situations.

3. *Macro Capability*: At each language level, except the lowest, a macro generation capability should be present. At the source language level this capability would provide for defining new operators and algebraic or logical subroutines. At the assembly language level this capability could be used to define often used instruction sequences in the usual digital computer programming manner, or often used component groupings on the analog computer. Thus for example the assembly language instructions which correspond to an amplifier, resistance network, pot and reference voltage might be combined in a macro to define an Integrate operator.

4. *Interactive Mode*: In order to maintain and enhance the interactive features naturally available on the analog part of the hybrid a run-time interactive mode should be present. Table II lists some of the functions which this mode should possess. Ideally run-time interaction should be permitted at each of the language levels. Although the interactive mode is not detailed in this paper, it appears to be a natural extension of the proposed programming system.

5. *Multiprogramming Capabilities*: At this time, no attempt is made to specify a general multiprogramming capability. A limited ability to assign problems to nonoverlapping sections of the analog patchboard is, however, desirable given the cost of such boards and the time required for patching. Allocation of components in this manner allows several small problems to remain wired on the patchboard at the same time while running only one of them in the hybrid configuration.

6. *Diagnostics*: One of the most useful features of digital programming systems has been the availability of extensive programmed diagnostics. These diagnostics aid the programmer in both compile time and run time program debugging. Similar capabilities must be present in hybrid programming systems if an effective system is to be developed.

In hybrid systems it is convenient to divide diagnostic functions into four types: syntax, structure, set up and dynamic. The first two types correspond to compile time diagnostics and the last two types correspond

TABLE II—Interactive mode functions

| ANALOG | DIGITAL |
|---|---|
| 1. Change parameter values | 1. Change parameter values related to utility routines |
| a. pot settings | a. integration step size |
| b. switch settings | b. function generation parameters |
| 2. Scale problem | 2. Change program defined parameters |
| 3. Obtain static check values | 3. Change program structure |
| 4. Perform static check | 4. Initiate and terminate run |
| 5. Control computer status | 5. Read out digital computer status |
| a. mode | |
| b. time scale | |
| 6. Change program structure | |
| a. patching change | |
| 7. Read out analog computer state | |
| a. Amplifier outputs | |
| b. Pot Settings | |

to run time diagnostics. Table III indicates some of the diagnostic capabilities associated with each type. The table is meant to be exemplitive rather than all inclusive and primarily defines a framework for diagnostic functions in hybrid systems.

It may be noted that there is nothing directly comparable to setup diagnostics in all-digital programming systems. This results from the assumption in all-digital systems that once the problem is represented correctly at any language level, a correct machine implementation will result. This assumption would also be correct for analog computers if an automatic patching device were available.[13]

7. *Documentation*: As in digital systems, documentation in hybrid systems can be divided into program and execution documentation. Program documentation includes general program listings, and for the analog part, wiring lists, pot settings, etc. Execution documentation is associated primarily with the interactive mode of the system. Thus, changing pot settings, run modes and the like are automatically documented when the change is requested. Structure changes such as adding an amplifier onto the patched problem should also be documented. These changes, however, will require separate entries into the digital computer representation of the analog problem in addition to performing the changes on the analog patchboard. This is necessary if a current picture of the analog patched problem is to be maintained in the digital computer. In addition, general run-time documentation of the digital programs currently being used, of amplifier outputs and pot settings, and general analog computer structure lists, should also be present.

8. *Special Analog Computer Reguirements*: Certain special compiler processing is required by the presence of the analog computer. Most of these special requirements were implemented on the APACHE system. They are listed briefly below:

a. Automatic Scaling: Automation of magnitude and time scaling is desirable. In addition, hybrid procedures for dynamic rescaling of hybrid problems to insure maximum accuracy should be provided.

b. Component Allocation: Components should be allocated automatically to the analog patchboard. The facility for assigning particular components or re-

TABLE III—Hybrid diagnostics

| ANALOG | DIGITAL |
|---|---|
| *Syntax* | |
| 1. Syntax Checking | 1. Syntax Checking |
| *Structure* | |
| 1. Connection symbol defined as input without being defined as output | 1. Program branches to nonexistent statement number |
| 2. Not enough components are available to implement program | 2. Program accesses nonexistent memory or subroutine |
| 3. An illegal configuration is encountered | |
| a. Logical signal feeding analog component | |
| b. An output connected to an output | |
| c. Reference voltage feeding function relay | |
| d. Pot feeding function generator | |
| *Set up* | |
| 1. Static Checking | |
| *Dynamic* | |
| 1. Overloaded amplifier | 1. Register overflow |

stricting the assignment of components to a particular part of the board, should be available.

c. Static Checking: An automatic static check procedure (setup diagnostic) is necessary which generates static check values and subsequently performs an on line consistency check between these values and those obtained from the patched problem.

d. Digital Simulation: Facility must be provided for digitally simulating parts of the problem which may eventually be implemented on the analog part of the hybrid. This is useful both in determining variable ranges for scaling purposes and in deciding which part of a problem is better implemented on the analog or digital subsection of the hybrid.

9. *Special Hybrid Requirements*:

a. Computer Allocation: Except for special cases, digital computer programming at the source language level does not usually involve the problem of computer resource allocation. For the analog computer programmer the allocation problem is confined to component allocation on the analog patchboard and is usually performed after the method of solution has been determined. For the hybrid computer programmer the question of allocation of computer resources can be viewed as one of the first questions to be answered. Very early in the problem formulation it must be determined which part of the problem is to be implemented on the analog subsection and which part on the digital subsection. This decision affects both the speed and accuracy of problem solution.

Assuming for the moment that criteria can be developed for determining the analog/digital implementation division of a problem, it is useful to consider two types of source languages. The first is for general problem representation independent of implementation, and the second indicates the implementation division between the analog and digital subsections.

Formulation of general computer allocation procedures are extremely difficult. It appears that in many instances the mathematical formulation of the problem contains certain prejudgments concerning the allocation problem. The area requires a good deal more study and is not considered further in this paper.

10. *System Implementation*: Experience with APACHE and other systems suggests that it is often desirable to sacrifice something in run-time program efficiency for the clarity gained by programming in a higher level language. The resulting program usually is more easily understood, and thus adaptable, by persons not intimately acquainted with the system. For the same reason system implementation should emphasize a high degree of program modularity.

*The programming system*

This section briefly discusses the language levels of a proposed hybrid programming system and the processing required between the levels. Figure 1 indicates the overall structure of the programming system. The programmer can enter the system by writing at the language level best suited to his needs. The language levels follow closely those defined for digital computer programming systems. The processing required between levels is outlined in Table IV.

1. *Hybrid Source Language (HSL)*:

In the Hybrid Source Language, the simulation problem is represented in a "natural" form, hopefully independent of its specific implementation. Such a language must have the ability to describe continuous systems and also have the general capabilities associated with higher level languages such as FORTRAN. CSSL (Continuous System Simulation Language[14]), a language developed by Simulation Councils Inc., has both of these capabilities. In addition, its macro and programmable structure features give it the great flexibility needed in a general simulation programming language. With some relatively small modifications, particularly in the control statement area, CSSL is taken here as the Hybrid Source Language.

In addition to syntax checking, the main processing at this level relates to the computer allocation problem discussed earlier.

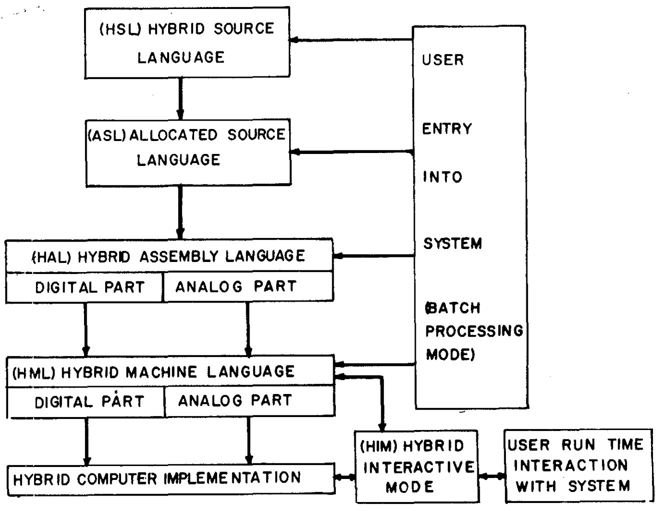

(HSL) HYBRID SOURCE LANGUAGE
(ASL) ALLOCATED SOURCE LANGUAGE
(HAL) HYBRID ASSEMBLY LANGUAGE
DIGITAL PART
ANALOG PART
(HML) HYBRID MACHINE LANGUAGE
DIGITAL PART
ANALOG PART
HYBRID COMPUTER IMPLEMENTATION
USER ENTRY INTO SYSTEM (BATCH PROCESSING MODE)
(HIM) HYBRID INTERACTIVE MODE
USER RUN TIME INTERACTION WITH SYSTEM

Figure 1—A hybrid computer programming system

TABLE IV—Processing between language levels

| Language Level to Level | PROCESSING OPERATIONS |
|---|---|
| 1(HSL) to 2(ASL) | 1. Syntax Checking<br>2. Allocation to analog and digital subsections of hybrid<br>3. Documentation |
| 2(ASL) to 3(HAL) | 1. Syntax Checking<br>2. Designation of variables as being generated on analog or digital subsections of hybrid<br>3. Allocation of interface channels<br>4. Establishment of necessary interface routines<br>5. Compilation of digital part of program<br>6. Compilation of analog part of program<br>a. production of analog assembly statements<br>7. Simulation of analog part, if requested<br>8. Execution of digital parts to obtain timing information, if requested<br>9. Documentation |
| 3(HAL) to 4(HML) | 1. Syntax Checking<br>2. Macro Processing and Listbuilding<br>3. Scaling Problem<br>4. Allocation of Analog Components<br>5. Production of Static Check Values<br>6. Documentation |

2. *Allocated Source Language (ASL)*:

The Allocated Source Language is effectively the same as HSL except that groups of instructions have been tagged to identify their implementation target. A modified form of CSSL may be used at this level. The main modification is in the addition of control statements ANALOG and DIGITAL which indicate the implementation of the block to follow. These statements may be combined with the CSSL block structure statements such as DYNAMIC and PROCEDURAL to form statements such as ADYNAMIC and DPRODCEDURAL which indicate that the statements in the dynamic and procedural blocks which follow are to be implemented on the analog and digital parts of the hybrid respectively.

Additional control statements are necessary to insure effective programming of the hybrid. Among these are statements specifying the analog output devices required, requests for a digital simulation of an analog implemented part of the program and statements indicating that lower level language instructions are to follow. A special control statement, TIME, is also necessary at this level to request timing information on digitally implemented blocks.

The main processing at this level involves the recognition of variables as being generated on the analog or digital part of the hybrid, the compilation of the separate analog and digital routines to produce assembly language versions of the separate parts and the establishment of interface routines to access and transmit those variables crossing the interface. It should be noted that compilation of the digital part of the program follows usual digital computer compilation procedures, the primary difference occurring when an analog generated variable value is needed. At this point either an interface utility routine is called to obtain the variable, or a table containing periodically updated variable values is acessed.

3. *Hybrid Assembly Language (HAL)*:

The analog oriented part of the Hybrid Assembly Language represents the analog part of the problem in terms of the physical components on the analog patchboard without being concerned with specific component allocation or component patching; e.g., the connection between an amplifier and a potentiometer is specified without indicating the specific amplifier and potentiometer on the patchboard. The digital part of HAL is just the assembly language associated with the digital subsections; it is not discussed further here.

Unlike the previous language levels, this language refers entirely to specific hardware components and as such is machine dependent. Currently such a language has been designed for use on the EAI 680 analog/logic computer. A translator for this language has been implemented in Fortran IV. The details of both language and translation are presented in the following section. The processing at this level is outlined in Table IV.

4. *Hybrid Machine Language* (*HML*): The analog oriented part of the Hybrid Machine Language is the same as the analog part of HAL except that each instruction now contains a specific number attached to it indicating the particular component on the analog patchboard on which this instruction will be implemented.

A user entering the programming system at this level can request all the processing associated with the previous level with the exception of component allocation which has already been performed.

*The hybrid assembly language*

This section discusses the details of the Hybrid Assembly Language. The section is divided into two parts. The first section describes Basic HAL, a symbolic analog interconnection language, and the second section describes Marco HAL, a macro language extension to Basic HAL.

1. *Basic HAL*: Basic HAL consists of a set of instruction types which enables the representation of analog components and their interconnections. The components are identified by mnemonic analog operation codes referred to as micro operators. Thus, for example, POTU designates an ungrounded potentiometer, while AMPL designates the amplifiers associated with the limit summer. A distinct micro operator is present for each analog and logic component available on the analog patchboard.

Connections between components are specified through the use of common connection symbols similar to a FORTRAN variable name. By using common symbols to represent the inputs and outputs of different components, connections between components are defined.

The general format for a Basic HAL instruction is:

(⟨output list⟩) = ⟨analog micro operator⟩(⟨input list⟩)

The output list is a list of connection symbols (termed output symbols) which associate names to the various outputs of the component designated by the analog micro operation. A list is necessary since certain components such as multipliers and relays have several outputs. The input list may contain connection symbols, parameter identifiers or numeric parameter values. The following example illustrates the use of four Basic HAL instructions in representing an integrator. The two inputs to the integrator are represented by X1 and X2, and the output by Y4.

```
(1)  Y1=REFP
(2)  Y2=POT(Y1,.5)
(3)  (,Y3)=RCNC(X1,,,X2,,,)
(4)  Y4=AMPC(,Y3,Y2,Y4,,,,,,)
```

The first instruction specifies a positive reference voltage which has the output symbol Y1. The second instruction is a grounded potentiometer set to .5, and fed by Y1. The output Y2 provides the initial condition for the combination amplifier specified in instruction (4). Instruction (3) specifies a resistance network associated with a combination amplifier. At the Basic HAL level this network must be specified since it is a component which can be used separately. The inputs to be integrated, X1 and X2, are fed into this network. The output of the network is fed into a combination amplifier using the common connection symbol Y3. Proper capacitive feedback is specified for the combination amplifier in instruction (4) by having the output symbol Y4 in the proper position of the input list. A dollar sign or blank is used to indicate that no input is present into these component positions.

Because of the large number of available components and their highly flexible interconnection, instructions at the Basic HAL level are often quite complicated. This complexity is, however, necessary at this level, if complete patchboard representation ability is to be achieved. The combination amplifier instruction is a good example of this complexity. Table V defines all the inputs to the combination amplifier in the notation employed in Reference 6.

In addition to the Basic HAL instructions several control statements are available. Among them are the PVALUE control statement which defines parameter identifiers and assigns numeric values to them, the

TABLE V—Combination amplifier instruction

Y = AMPC (X1, X2, X3, X4, X5, X6, X7, X8, X9, X10)

| | |
|---|---|
| X1: | AJ (Amplifier Junction) Input |
| X2: | OJ (Operate Junction) Input |
| X3: | IC (Initial Condition) Input |
| X4: | F (Integration Feedback) Input |
| X5; X6: | Mode Control Inputs |
| X7; X8: | Feedback Capacitor Selection Inputs |
| X9: | IJ (Initial Condition Summing Junction) Input |
| X10: | C (Feedback Capacitor Disconnect) Input |

HALEND control statement which terminates the HAL program and the ALOCATE control statement which is described below.

The programmer can specify the use of particular components on the patchboard by appending the analog micro operator with a component number. Thus writing

Y2 = POT 02 (Y1,.5)

will direct the compiler to allocate potentiometer 02 when implementing this instruction. Instructions in which particular components are specified are effectively analog machine language instructions. Due to this a separate discussion of the analog machine language is not necessary. The capability is also provided for restricting component allocation to certain parts of the analog patchboard. This is done through use of the control statement ALOCATE. ALOCATE is followed by number pairs which designate sequences of trays from which components are to be selected for problem implementation. On the EAI 680 trays are basic modules into which the board is divided; one tray may contain one or more components. Thus the statement ALOCATE (000,029) would restrict problem implementation to the first thirty trays of the EAI 680 patchboard. Naturally if it is not possible to find the appropriate components or number of components in the trays specified a diagnostic message is printed out.

2. *Macro HAL*: The Macro–HAL language consists of Basic-HAL with additional procedures for generating macros. In addition, a standard set of system implemented macro instructions is provided for certain commonly used instruction groups. A macro instruction is generated by supplying the assembler with a macro definition. The Macro HAL instruction format is the same as the Basic HAL format except that in addition to the analog micro operator there is an analog macro operator.

(<output list>) = <analog macro operator> (<input list>)

This analog macro operator is defined in the macro definition and is any identifier not already used as a micro or macro operator. To call a macro, one simply writes the Macro HAL instruction with the appropriate macro operator, connection symbols and parameters in their places.

Table VI indicates the format required in a macro definition. A header, AMACRO, and a trailer, AMEND, define the beginning and end of the macro definition. The prototype statement is the Macro HAL instruction in the format given above. This defines the macro operator and the number and position of the inputs and output symbols to be expected. The next statements required are declaration statements. These macro control statements indicate which identifiers are to be considered as connection symbols and which as parameter identifiers. The body of the

TABLE VI—Macro definition format

| | |
|---|---|
| AMACRO | |
| "Prototype Statement" | |
| "Declaration Statements" | |
| "Body" | 1. Macro HAL Instructions |
| | 2. Macro Assembly Instructions |
| AMEND | |

program consists of Macro HAL instructions described earlier, and Macro Assembly instructions. The Macro Assembly instructions provide for symbol operations such as substitution, arithmetic operations on parameter values and identifiers, and conditional operations for expansion time component and parameter changes. With these facilities, very flexible macros can be written which conditionally adapt the implementation structure to the requirements of the problem.

Most of the major operators such as integrate, sum, etc., associated with digital simulation languages and requiring several components for analog implementation are provided as system macros. These assembly language system macros also represent a target language into which the differential equation based notation of the Allocated Source Language is to be translated.[18]

*Processing the hybrid assembly language*

The processing required for HAL is indicated in Table IV. The first tasks of syntax checking, macro expansion and list building result in the production of a linked list and several associated tables. Together they represent an easily accessed and processed internal digital representation of the analog problem.

Scaling the problem and producing static check values, though not yet implemented, also occur at this level. These tasks may be performed at higher language levels, however, this level has been chosen to facilitate rapid on-line programmer interaction. Thus once changes have been made in the structure of the analog problem on the patchboard, the equivalent changes can be made in the internal digital computer representation of the problem and new scaling and static check information can be requested.

Allocation of components to the analog patchboard is automatically done at this level. When performed manually, the allocation involves matching components to blocks in a block diagram problem solution. The criteria for such assignments are often qualitative and include such notions as compactness of patchboard wiring and neatness or symmetry in wiring appearance, both of which aid in problem debugging. The assignment itself, though sometimes tedious, is easily effected using these visual qualitative criteria.

When mechanizing the component allo ation task on the digital computer, two main approaches are available. The first attempts to give meaning to qualitative criteria such as compactness and neatness through the development and subsequent optimization of appropriate objective functions. Objective functions such as wire length, wire crossovers, and area covered on the board are often used in digital computer backboard wiring and, to some degree, do reflect the concept of compactness. The general problem of allocating objects (components) to locations on a board, subject to restrictions on object placement, with the goal of minimizing some objective function is often referred to in the literature as the "assignment" or "placement" problem.[15] Algorithms for the solution of such optimization problems can take several hours of computing time[15] when several hundred objects and locations are present. This is due largely to the astronomical number of ways one can allocate a given problem and the slow and not easily predicted convergence properties of available algorithms. The large computing time requirements make this approach unsuitable for a short compile time or on line programming system. In addition, it is not clear that these objective functions meaningfully quantify the qualitative allocation criteria generally employed by programmers.

The second approach to the component allocation task is to develop a set of heuristic algorithms which try to embody concepts such as compactness and neatness while at the same time keeping computing costs at a minimum. As with most heuristic algorithms, the one currently implemented in this system has worked well on most, but not all of the problems it has encountered. In every problem it does, however, find a legal allocation if it is possible. The basic assumptions of the heuristic are given below.

a. Components used as integrators and summers are generally the key elements in determining the way a programmer patches a board, or draws a flow diagram. These components should therefore be allocated to preserve, as much as possible, the visual signal flow patterns between them.

b. Patching situations such as initial condition pots and pots tied to the inputs or output of amplifiers should be considered as special cases. In many of these and other cases, the patchboard of the EAI 680 has been designed for neatness and compactness by providing special patchplugs which may be used instead of wires. Since plug patching is both neat and represents a minimum wire length it should be utilized where possible.

c. A certain amount of patching compactness is desirable. The remainder of the components should therefore be allocated in the basis of their closeness to already allocated components.

These assumptions form the basis of a three phase allocation algorithm with each phase corresponding to one of the assumptions above. The details are not discussed in this paper. The example provided in the next section, however, demonstrates the results of the algorithm.

*An example*

Figure 2 contains the scaled block diagram of an automobile suspension simulation.[16] Table VII is an input listing of the problem as represented in the Basic HAL language. The same problem represented with the use of the system macros integrate (INTG), summation (SUM), and invert (INVT) is given in Table VIII. There is approximately a three to one reduction in code lines required when the problem is represented using the macro facility and this representation is reasonably clear and compact. The resulting allocation of the problem to the EAI 680 patchboard is given in Figure 3 and indicates that much of the problem's visual signal flow patterns have been preserved.

The programming system which to date includes the syntax checking, and the macro, listbuilding and allocation processing described previously has been implemented in FORTRAN IV. For the example above, this processing took approximately four seconds when executed on a Univac 1108 computer. A more complex problem, the Cable Arrestor Problem,[17] containing roughly twice as many components took nine seconds.

## CONCLUSIONS

This paper proposes a hybrid programming system in terms of four language levels and the processing required between them. Some of the details of the lowest language levels which have been implemented are presented and an example demonstrating the use of the system is given.

Currently the authors are engaged in completely specifying the modifications necessary for transforming CSSL into a desirable allocated source language. A continuing study is also being made of the interface requirements between the analog and digital program subsections. In addition implementation continues on the lower level processing tasks, the initial goal being a subsystem which handles the analog subsection of

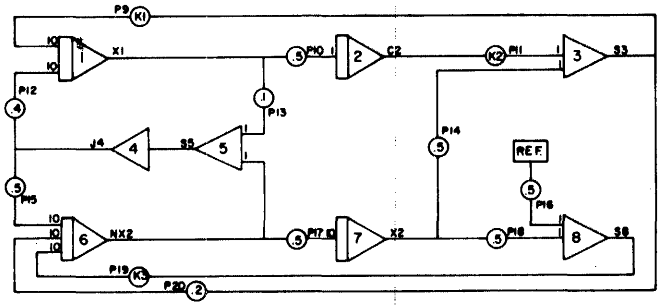


Figure 2—Simulation of an automobile suspension system

TABLE VII—Basic HAL input for automobile suspension problem

```
PVALUE (K1 = .16, K2 = .5, K3 = .8)
P9 = POT(S3, K1)
P10 = POT(X1, .5)
P11 = POT(C2, K2)
P12 = POT(J4, .4)
P13 = POT(X1, .1)
P14 = POT(X2, .5)
P15 = POT(J4, .5)
P16 = POT(NREF, .5)
P17 = POT(NX2, .15)
P18 = POT(X2, .5)
P19 = POT(S8, K3)
P20 = POT(S3, .2)
(,R1) = RCNC(,,,P9, P12,,)
(,R2) = RCNC(P10,,,,,)
(,R6) = RCNC(,,,P15, P20, P19,)
(,R7) = RCNC(,,,P17,,,)
R3 = RCNS(P11,P14,,,,,S3)
R8 = RCNS(P16,P18,,,,,S8)
R5 = RCNS(P13,NX2,,,,,S5)
J4 = AMPJ(S5,,)
X1 = AMPC(,R1,,X1,,,,,,)
C2 = AMPC(,R2,,C2,,,,,,)
NX2 = AMPC(,R6,,NX2,,,,,,)
X2 = AMPC(,R7,,X2,,,,,,)
S3 = AMPS(R3,,,,,)
S8 = AMPS(R8,,,,,)
S5 = AMPS(R5,,,,,)
NREF = REFN
HALEND
```

TABLE VIII—Macro HAL input for automobile suspension problem

```
PVALUE (K1 = .16, K2 = .5, K3 = .8)
X1 = INTG(K1,S3,.4,J4)
C2 = INTG(.,5X1)
NX2 = INTG(.5, J4, .2, S3, K3, S8)
X2 = INTG(.5,NX2)
S5 = SUM(.1,X1, 1, NX2)
S3 = SUM(K2, C2, .5, X2)
S8 = SUM(.5, X2, 5, NREF)
J4 = GAIN(S5,1)
NREF = REFN
HALEND
```

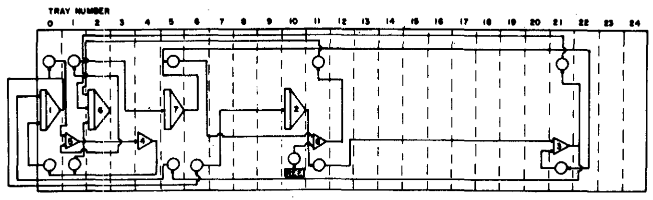


Figure 3—Automobile suspension problem allocated to EAI 680 patchboard

a program at the Macro HAL level, and contains a limited interactive mode (Figure 1) capable of online scaling and static checking in response to patchboard configuration changes.

Programming costs for hybrid computers have mushroomed to the point where the economic justification of hybrid simulation projects is being questioned. It is hoped that this proposal will both stimulate discussion in this area and fill a current and growing need for an effective hybrid programming system.

## REFERENCES

1 G GORDAN
*GPSS-A general purpose systems simulation program*
IBM Systems Journal Vol 1 1962 18-32

2 H M MARKOWITZ B HAUSNER H W KARR
*SIMSCRIPT: A simulation programming language*
Prentice-Hall Inc N J 1963

3 J J CLANCY M S FINEBERG
*Digital simulation languages: A critique and a guide*
Proc FJCC Vol 27 1965 23-36

4 J C STRAUSS
*Digital simulation of continouus systems: An overview*
Proc FJCC Vol 33 1968 339-343

5 T D TRUITT
*Hybrid computation . . . What is it? Who needs it?*
IEEE Spectrum Vol 1 No 6 1964 132-146

6 EAI #680 Reference Handbook
Electronic Associates Inc N J 1967

7 C GREEN H D'HOOP A DEBROUX
*APACHE—A breakthrough in analog computing*
IRE Trans on E C Vol 11 1962 699-706

8 *APACHE: Analog programming and checking programmers manual*
Euratom Doc EUR 2437 e 1966

9 *APACHE: Analog programming and checking system programmers guide*
Euratom Doc EUR 3052 e 1966

10 W OCKER S TEGER
*HYTRAN—A software system to aid the analog programmer*
Proc FJCC Vol 26 1964 291-298

11 W MIESSNER
*APACHE Subcommittee report*
SCI Simulation Software Committee 1965

12 J KOVACS J C STRAUSS
*An approach to a hybrid programming language*
SCI Third Annual Simulation Software Meeting 1967

13 T J GRACON J C STRAUSS
*A decision procedure for selecting among proposed analog computer patching systems*
Simulation Vol 13 No 2 1969

14 J C STRAUSS editor
*CSSL—The SCI continuous system simulation language*
Simulation Vol 9 No 6 1967

15 M BREUER
*Design automation of digital computers*
Proc IEEE Vol 15 1966 1700-1720

16 *EAI handbook of analog computation*
Electronic Associates Inc N J 1967 Chapt 3 119

17 A E ROGERS T W CONNOLLY
*Analog computation in engineering design*
McGraw-Hill Co Inc 1960 379

18 M STEIN
*Automatic digital programming of analog computers*
IEEE Trans on E C Vol 12 1963 100-111

19 H PAYNTER J SUEZ
*Automatic digital set-up and scaling of analog computers*
ISA Trans Vol 3 1964 55-64

# Hybrid executive—User's approach

*by* W. L. GRAVES and R. A. MacDONALD

*TRW Systems Group*
Redondo Beach, California

## INTRODUCTION

Hybrid executive programs have long been prevalent in the hybrid computer simulation industry, however, what should be the essential features of a hybrid executive is still a controversial subject. For the most part, the design of hybrid executives has been undertaken by the manufacturers of hybrid systems and in many designs the complexity in the operation of these programs has resulted in their usage only on large class digital systems. Consequently, hybrid facilities which employ a small to medium class digital computer system are faced with the task of developing an executive program compatible with the facility environment. However, in many of these small to medium hybrid facilities, the segregated program development effort for a hybrid executive is not undertaken until considerable time after the installation of the hybrid system. The normal reasons are inadequate programming funds or a higher priority assignment of available personnel to satisfy programming and development needs of existing hybrid simulations.

For hybrid computation, specifications for the executive design must include sufficient flexibility to enable the user to easily alter the mode of the executive execution at run time as well as at compilation time to meet the requirements of the particular engineering problem being simulated. In hybrid executives existing today, such flexibility does not generally exist. These executives usually consist of a conglomeration of many programs that perform specific functions and are linked together only to the extent that the order of their execution is controlled by a simple monitor. However, the nature of these functions is such that the provision of linkage between control and problem data could considerably reduce the complexity of their implementation while increasing flexibility.

In this paper, the philosophy for a hybrid executive design, which has evolved from extensive user experience, is described. Since it is a user philosophy, it is relatively unique in the hybrid simulation industry wherein most designs are specified by "software experts", which usually have attained their expertise via an all digital environment. A definition of the term "user" is in order. A user is defined as a person in the role of either an applications programmer or engineering analyst as opposed to a system software programmer or analyst. The hybrid executive (hereafter referred to as the TRW executive) discussed in this paper was primarily developed to satisfy the simulation requirements for a large aerospace engineering problem. However, the authors feel that the extended usage of this executive to other applications, whatever the size, is reasonable, The general requirements for this problem and the rationale used in the design of the executive programs are discussed.

### *Typical executive requirements for hybrid simulation*

In early 1967, the TRW Analog/Hybrid Facility had been requested to develop a large multi-use hybrid simulation capability in support of the Apollo program. For this study, which involved several independent simulations, each basically simulating two vehicles in 6 DOF and employing as many as two control systems for each vehicle, it became very apparent that total executive control for each of these simulations would be required for the following reasons:

- The size and complexity of the simulations would require an extensive daily checkout to assure simulation readiness. To accomplish this task by manual means on the analog would be impractical, and therefore, potentiometer setup and static checkout using digital control would be required. Also, since it was expected that the definitions of the simulation state would change frequently, either due to changes in parameters or to different selections of program options, the pot setup and static checkout programs should have sufficient flexibility to assure analog or system readiness for the current simulation definition.

- Complete flexibility in the data input and output formats, such that either the simulation staff or the various engineering analysts assigned to this project could communicate with the simulations in a familiar, user oriented, language and without burdened details of specific data formats.
- A large simulation staff of programmers of varying experience and backgrounds would be assigned to the program, therefore, generalized software to handle control such as interrupts, analog/digital interface, sampling, etc., need be developed such that program interfacing would not be a difficult task.
- Because of the size and complexity of the simulation and because of an additional requirement to be able to use the simulations for a multiple of studies, scaling of both amplitude and time would be difficult to specify prior to execution. Therefore, the capability to rescale at run time would be necessary to reduce considerably the recompilations required if this information is fixed within the program.
- A requirement to display the dynamic status of up to several hundred variables either digitally and/or via the analog would be necessary. Because digital display using a line printer during problem execution would be time prohibitive, a dynamic dump capability to external bulk storage (disc drives or magnetic tapes) for later recovery or further processing would be required.
- Because of the potential multiple of uses for the simulation programs, data I/O requirements from study to study would be expected to vary considerably. Since it would be highly inefficient to recompile the programs for each new I/O configuration, the executive capability must include a means for defining the I/O processes at execution time rather than at compilation time.
- Since the total digital program storage requirements were expected to exceed available memory, the executive program structure must provide capability for program overlay and data interfacing in a manner not overburdening to either the user or the respective programmers.

In satisfying the requirements for executive control of the Apollo simulations, two important constraints were applied. First, development and design effort of the executive must be done within the budget and schedule allotted by the Apollo simulation task, and second, sufficient generalization and compatibility must be maintained in the design for adaptation to other digital software systems, if necessary, during the simulation effort. This latter constraint implies that the design and implementation should not require modification of software provided by the computer manufacturer, (loader, compiler, I/O, etc.) for operation.

*Evolution of the executive design and development*

In the Hybrid Computation Facility at TRW Systems Group, which currently employs a medium class digital computer (CDC-3100) linked to four analog computers (two Beckman 2132's and two Comcor CI-5000's), a generalized hybrid executive program was not available for nearly three years from the time of installation in 1964. A reasonable software development activity within TRW could not be initiated with the available personnel because of committments to simulation development for several large programs. Prior to late 1967, executive control for hybrid simulations was tailored specifically to fulfill the requirements for the particular study and was generally not applicable from study to study. However, valuable experience had been gained in realizing, from a usage point of view, the total requirements and capabilities for a generalized hybrid executive program.

Upon the initiation of the Apollo simulations in 1966, two approaches for developing a hybrid executive were considered. One approach was to develop a complete executive separate from the problem implementation and later integrate the two programs for final checkout. A second approach was to develop the executive in parallel with the problem implementation and integrate and check out the combined modules of the simulation as they were developed. From the stringent Apollo simulation schedule, it was apparent that the latter approach would be more feasible. Consequently, the design evolution of the executive was dictated by satisfying the particular simulation requirements at the time of implementation. As a result, many of the capabilities presently existing in the TRW executive have resulted from second or third generation design changes as user flexibility and program efficiency so required.

*Program description*

Several basic philosophies were adhered to during the executive design and development:

1. Any information required in defining the simulation which may change frequently is entered as data at run time. This class of information includes items such as scale factors, linkage assignments, analog component or console assignments, required program sequencing control flags and all problem parameters.
2. Any information that is changed only if the engineering system being studied is redefined is compiled into the system. This would include items such as problem equations, etc.
3. All control or problem executions which are non-time critical, that is, not required for the dynamic execution of the problem, need not reside in memory during the time critical execution. Functions such as pre-data and post-data processing, initialization, pot value determination and setting, static check determination and interrogation are non-time critical and are usually executed once per run sequence and therefore may be program overlayed, thus optimizing or reserving resident core for the time critical or "Real Time" program.
4. All data values required to transfer information or problem status between major program functions must reside in core using a "COMMON" reserved data area. It is this important constraint on implementation that permits the usage of program overlaying and aides significantly in the executive design.

Five or six separate computer functions or programs can be defined, which satisfy the total simulation requirements: data I/O processing, initialization, pot evaluation and setting, real time execution, static check evaluation and interrogation, and possibly, post data processing. Figure 1 depicts the general organization of these functions. It should be noted, that the order of execution of these functions is completely determined by the user at run time from data input, and that any single function can be executed separately or by an automatic sequencer.

Since overlaying processing is used, each function or program comprises, but not necessarily so, a separate computer overlay with each in turn further overlayed (with the exception of the real time program) as increased core requirements are experienced. Each of these programs is executed by a simple driver or monitor upon command by the user utilizing the resident COMMON for data transfer. In the following sections, the design for each of the five major programs and the control of their execution is briefly discussed.

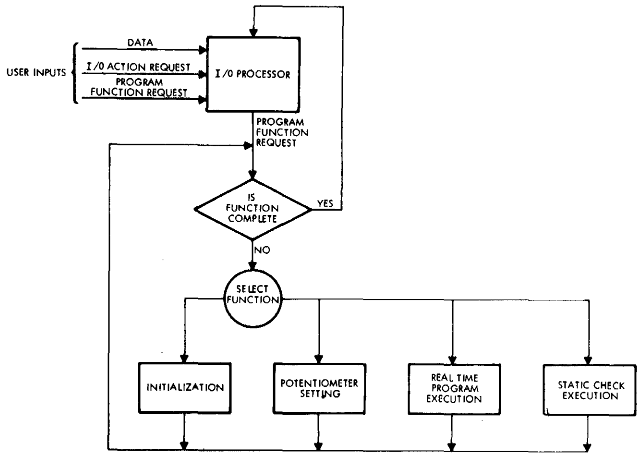


Figure 1—Hybrid executive program structure

## Data I/O processing

Because the most frequent interaction between the user and the system occurs through the I/O portion of the executive, special attention is warranted to make the interaction as painless as possible. Since the external characteristics of entering both data and action requests are identical for the TRW executive, the following comments generally apply to both classes of information.

The essential task performed by I/O software is the conversion between data representations required externally to the computer. Each time an item of information is processed for I/O, a description of the item sufficient to allow conversion must be available. The TRW executive requires inclusion of descriptors that specify the following. Names entered must be defined as data identifiers or action requests identifiers. The internal classification of the data, REAL, INTEGER, OCTAL, etc., must be specified. Differentiation must be made between data that is part of an array and data that is not. Conversion from one set of engineering units to another is also allowed and must be specified.

Clearly, any I/O format that requires specification of all of these descriptors every time an item is referenced is untenable. In the TRW executive, the approach used to reduce the problem requires the user to provide a list of all names that are to be accepted and the required descriptors of each. Specification of the required descriptors is done using FORTRAN oriented names such as REAL, INTG, etc. This list is compiled to allow ease of linkage with appropriate I/O handling routines. Once the list is defined, entry of data requires only a name and a numeric value. Since all conversion is pre-specified, no artificial indicators, such as a decimal point to specify a floating point number, are required. Since it is reasonable to expect the descriptors defined for each data value will not change unless the problem definition changes, no appreciable loss in flexibility for I/O processing is realized when the descriptor list is compiled.

The internal definition of conversion requirements also permits extremely simple definitions of display requirements. In this case, the data value already exists within the computer and only the name of a variable is necessary to complete the information needed within the computer to define output requirements. Indirectly, this has allowed requesting all display functions by simply entering a list of names. The implications contained here are best illustrated in the case of specifying "Dynamic Dump" requirements. This is an output function that should be time optimized. Unfortunately, optimization of a routine to output floating point data requires different instructions than those needed for output of fixed point variables. In view of this, a problem arises when it is desired to intermix floating and fixed point numbers in a single general request list. The TRW executive, since it has access to all pertinent descriptive information, can handle this problem internally without the user even being aware that it is happening. The allowance of such mixed mode lists is provided for printing and dynamic dumps.

Another problem often encountered in trying to enter data into a computer is caused by the presence of rigid format structures such as requiring that items be aligned to specific card columns. Where users are often required to hurriedly keypunch or type in their own data for performance of runs, such rigidity becomes too restrictive. Thus, one design criterion for the I/O package was the elimination of this problem. A solution was achieved through use of an input string scanning routine which searches an entire input record for appropriate data fields.

In the case of action requests two forms exist and are distinguished only by their manner of use. The first form, which is the larger class, is referred to as an I/O action request and the functions performed are restricted to various manipulations of data. Requests for saving program status on a disk or transferring data from cards to tape are examples. Basic to this class is the requirement that the subroutine used to process the request returns control to the executive input output controller. The second class of action request, referred to as program execution requests, is used to initiate execution of hybrid functions not related to I/O. In this case the routine used to satisfy the request passes control to the executive execution sequence controller rather than the I/O controller. In both cases, the specification of the request to the executive program is the same and the user implies through his own subroutine the class to which the request belongs. Figure 2 shows the control used for I/O processing in conjunction with how this control interfaces with the executive control of those functional blocks as indicated in Figure 1.

## Potentiometer evaluation and setting

As part of performing each and every computer run, potentiometers must be set to the proper values. In most small and medium sized hybrid labs the ability to do this from the digital is provided with one of two levels of sophistication. The first requires specifying the address of the potentiometer to be set and the value to which it must be set. The second requires specification of the potentiometer address, parameter values and a FORTRAN like expression used in computing the setting, The latter then both computes the

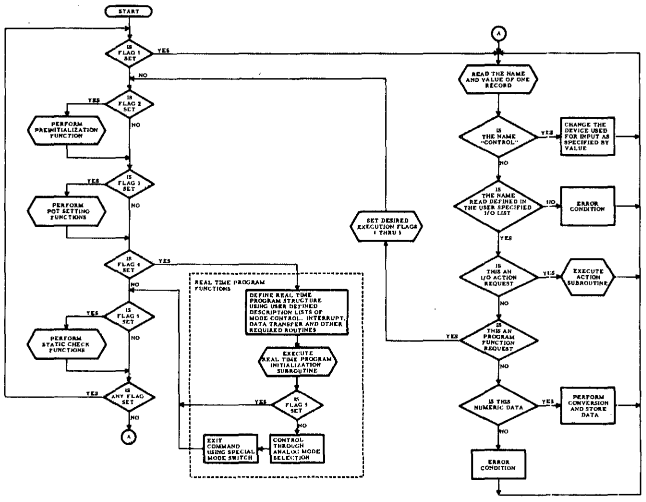

START
IS FLAG 1 SET
YES
NO
IS FLAG 2 SET
PERFORM PREINITIALIZATION FUNCTION
IS FLAG 3 SET
PERFORM POT SETTING FUNCTIONS
IS FLAG 4 SET
IS FLAG 5 SET
PERFORM STATIC CHECK FUNCTIONS
IS ANY FLAG SET
A
REAL TIME PROGRAM FUNCTIONS
DEFINE REAL TIME PROGRAM STRUCTURE USING USER DEFINED DESCRIPTION LISTS OF MODE CONTROL, INTERRUPT, DATA TRANSFER AND OTHER REQUIRED ROUTINES
EXECUTE REAL TIME PROGRAM INITIALIZATION SUBROUTINE
IS FLAG 5 SET
EXIT COMMAND USING SPECIAL MODE SWITCH
CONTROL THROUGH ANALOG MODE SELECTION
SET DESIRED EXECUTION FLAGS 1 THRU 5
READ THE NAME AND VALUE OF ONE RECORD
IS THE NAME "CONTROL"
CHANGE THE DEVICE USED FOR INPUT AS SPECIFIED BY VALUE
IS THE NAME READ DEFINED IN THE USER SPECIFIED I/O LIST
ERROR CONDITION
IS THIS AN I/O ACTION REQUEST
EXECUTE ACTION SUBROUTINE
IS THIS AN PROGRAM FUNCTION REQUEST
IS THIS NUMERIC DATA
PERFORM CONVERSION AND STORE DATA
ERROR CONDITION

Figure 2—Executive and I/O processing control

setting and automatically sets the potentiometer using an interpretive compiler. Both of the methods require that the user select those potentiometers whose settings will change. This selection is based on the engineer's knowledge of parameter value changes, and in lumped parameter definitions or where the same parameter is used repeatedly throughout the problem, this can be very cumbersome.

The TRW executive automatically includes the necessary setting changes in the digital program and thus relieves the user of an unnecessary burden. Since the actual setting is the only number associated with a potentiometer that reflects parameter variations, it is used to initiate resetting of potentiometers. The method used is as follows: a list containing all setting values is retained on bulk storage; as part of each run, all potentiometer settings are computed and compared to the list; a difference between the two values automatically results in a resetting of the potentiometer and the list being changed to reflect the new value.

Although the concept used is very simple, there are implications that markedly affect program implementation. The most pertinent of these is the requirement that all current parameter values be available to the program which computes the potentiometer settings. To easily make these values available and to still retain the speed necessary to make computation of all settings feasible, requires compilation of the setting evaluation routine instead of using an interpretive routine as do many of the hybrid computer manufacturers. Clearly, interpretive methods offer considerable flexibility in specifying potentiometer values, but the authors believe that this degree of flexibility is not necessary

Before clarifying this point of view, a definition is in order.

Let $P_S = A_{SF} \cdot D_P$

where $P_S$ = Potentiometer setting

$D_P$ = Pot definition

$A_{SF}$ = Analog scale factor

Assuming the reader is familiar with the meaning of "potentiometer setting" and "analog scale factor" the given equation will suffice to define "Pot Definition". The important characteristics of a pot definition are its dependency only on physical parameter values and its corresponding independence of potentiometer address or analog scale factor.

Dependency only on physical parameters implies that a "pot definition" changes only when the problem being solved is redefined in a manner such that equations are changed, which in most simulations is relatively infrequent. Thus, if the "potentiometer setting" program requires recompilation only when "pot definitions" are changed, no significant loss of flexibility is encountered.

As a result of these considerations, the routine was formulated such that analog scale factors and potentiometer addresses were entered as data and "pot definitions" were coded into a FORTRAN subroutine Use of this method utilizes the full capability of COMMON while retaining the flexibility at run time in specifying values (scale factor, component address) most likely to vary.

Since the program (Figure 3) necessary to compute the actual setting (i.e., form the product of the "pot definition" and the scale factor), compare old and new values and handle bulk storage files is the same for any problem, it is formulated as part of the executive. Definition of the pot setting requirements for a given problem consists of coding the FORTRAN list of "pot definitions" and preparing the list of pot addresses and scale factors. The analog data associated with a pot is stored in a serial file on a disk. This data consists of the pot address, analog console number, analog scale factor, present value of pot setting, and an index

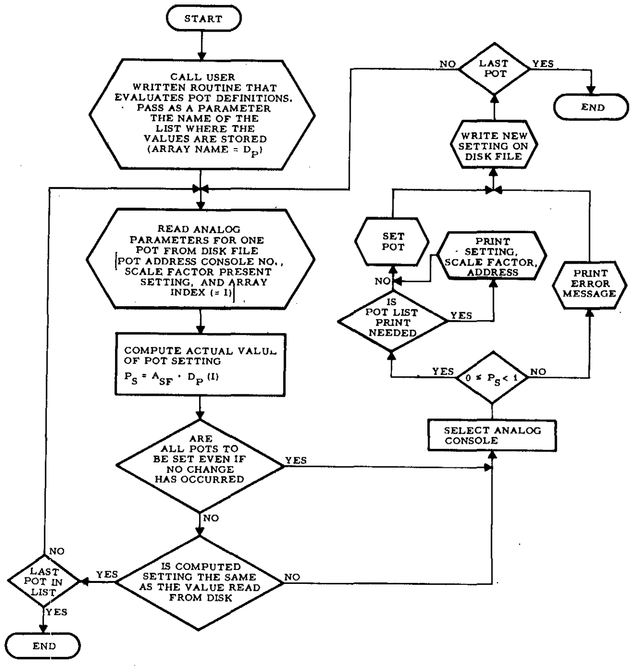

START
CALL USER WRITTEN ROUTINE THAT EVALUATES POT DEFINITIONS. PASS AS A PARAMETER THE NAME OF THE LIST WHERE THE VALUES ARE STORED (ARRAY NAME = $D_P$)
READ ANALOG PARAMETERS FOR ONE POT FROM DISK FILE (POT ADDRESS CONSOLE NO., SCALE FACTOR PRESENT SETTING, AND ARRAY INDEX (= I))
COMPUTE ACTUAL VALUE OF POT SETTING $P_S = A_{SF} \cdot D_P(I)$
ARE ALL POTS TO BE SET EVEN IF NO CHANGE HAS OCCURRED
YES
NO
IS COMPUTED SETTING THE SAME AS THE VALUE READ FROM DISK
YES
NO
LAST POT IN LIST
NO
YES
END
SELECT ANALOG CONSOLE
$0 \leq P_S < 1$
YES
NO
IS POT LIST PRINT NEEDED
YES
NO
SET POT
PRINT SETTING, SCALE FACTOR, ADDRESS
PRINT ERROR MESSAGE
WRITE NEW SETTING ON DISK FILE
LAST POT
NO
YES
END

Figure 3—Potentiometer setting control

(I) which defines where in an array ($D_P$) the value of the pot definition is stored by the FORTRAN routine used to evaluate the pot definitions.

### Initialization or finalization

In engineering simulations, the analyst prefers to have the mechanization in a form which is either familiar to him or closely related to the physical system being simulated. In digital simulation, the analyst is usually far removed from the program, and if his results are of a suitable form, the actual formulation of the equations is of little interest and can therefore be optimized for computer efficiency and stability.

To the contrary, in analog or hybrid simulation where a close rapport with the program is desirable, mechanization in either an optimum or in a less computer sensitive manner is often traded off against a more realizable formulation. As an example, an analyst might have access only to data determined in a reference frame that differs from the reference frame best suited for use within the computer (e.g., gimbal angles vs direction cosines). In such cases, reformulation of values for computer initialization may require extensive computation. In hybrid computer simulations the digital computer can be used to determine such values regardless of the complexity. With this capability, the total simulation can be formulated for optimum execution, and often better computer stability, and the results transformed to the users preference without decreasing the flexibility to the user or analyst.

To accomplish the reformulation transfer from the user desired input form to the program execution form to the user desired output form, non-time critical digital calculations, which may be considerable, need be performed. Examples would be coordinate transformations, root extraction, curve fitting, data analysis, etc. Since these types of calculations are executed only once each run cycle, they can be programmed using FORTRAN, extended precision, and non-optimal programming techniques with a negligible increase in the system execution or throughput. It is this purpose that the preinitialization and/or finalization programs serve. Since these programs are entirely dependent on the problem being simulated, the only executive function is the call to these programs and the provision for data linkage through the use of COMMON.

### Real time program

The specification of software that would appreciably aid in getting the real time program operational was based on a generalization of the kind of problem that would be solved using the system. It was assumed that the physical system being studied could consist of several interacting subsystems each having a unique frequency content. For example, in the Apollo studies the kinematics and dynamics are frequency separable. This kind of system implied a computer program consisting of several loosely interacting subprograms each having its own timing and sampling requirements. Two primary questions to be answered were "What can an executive do that will provide assistance in programming each subprogram?" and "What aid may be provided in correlating the subprograms to represent a complete system?"

Two facts immediately suggested general answers to the questions above. Because each simulation represents a different system, the equations solved in the real time program are essentially unique for each new problem, and can be considered only by the user. At the same time, certain functions such as mode control and inter-computer data transfer are common to all simulations and characteristically depend only upon the computer system being used. Experience has shown that the user normally displays considerable ability to solve problems associated to his equations, but that his performance deteriorates markedly when dealing with computer system dependent functions. Clearly, a general executive can only address itself to aid in handling the computer system dependent problems present in the real time program. It is also clear, however, that these are the areas where the user most needs aid.

The TRW executive includes three major activities within the real time part. It provides generalized software to handle ADC/DAC specification control of the "dynamic dump" (time histories), and mode control and interrupt processing. Relating these activities to the questions above, it is found that generalization of these functions provides assistance to the user both in programming individual sub-programs and in overall system correlation. Justification of this last statement requires a more detailed description of each of the functions considered and their interaction with the user.

Although extensive details pertaining to the methods used in implementing the TRW executive are not appropriate, some indication of the gross approach used is appropriate. The available interrupt structure allows execution of up to eight concurrent real time subprograms (this limit of eight is caused by the maximum number of programmable interrupts available in the TRW hybrid system). A subprogram naming convention has been adopted to allow flexibility in choosing the interval at which variables are stored on bulk storage or at which variables are transferred for display purposes. The subprograms are arbitrarily named LOOP1, LOOP2, etc., up to the maximum number allowed by the interrupts available. The number associated with the loop is then used as a key to initiate certain action. In the case of dynamically dumping variables, the following scheme is used: each subprogram includes a call to the routine which performs the dump operation; the parameter passed with the call is the loop number; this number is compared to a number entered as data which specifies the subprogram, and thus, the time interval at which the dump is to be made; if the numbers compare, a dump occurs. A similar system is used for selecting inter-computer display transfers.

Associated with each subprogram are the following parameters which may be entered as data:

Present problem time
Time interval at which the subprogram is executed
Address of the first ADC channel used
Number of ADC channels used
Address of the first DAC channel used
Number of DAC channels used
Interrupt priority level

This data is stored as blocks in a predefined order known to each executive subroutine used in the real time program. Such a block structure permits usage of the same calling sequence to execute all executive subroutines, thereby reducing the chance of programmer error to a minimum.

## ADC/DAC specifications

The handling of ADC/DAC specifications within a program would seem to present little difficulty since even the most sophisticated DAC or ADC routine should require no more than three or four parameters. However, in many systems, specification of these parameters requires compilation. Such a requirement not only removes flexibility by requiring recompilation to incorporate changes, but also forces the assignment of specific equipment in a relatively early stage of program development. At the time a particular subprogram is written, it is usually not convenient to assign specific ADC's or DAC's since requirements for all subprograms must be considered in determining the best distribution. Similarly, conversion scale factors may change at any time. Another capability convenient for the user is flexibility in specifying inter-computer data transfer for purposes of display. This requires specification of specific DAC's or ADC's, the variables to be transferred, the scale factors to be used, and the time interval at which the transfer occurs. In view of these considerations, it seems reasonable to require software that allows assignment of all parameters associated with ADC's and DAC's at run time.

The actual assignments are made by entering two lists of data; the first containing the names of the variables to be transferred, and the second containing the conversion scale factor. The lists are entered in an order corresponding to the ADC or DAC line that is being used. In the I/O processor, the list of names is replaced by a list of the addresses of those names and this along with the scale factor list, is passed to the real time program for tailoring of the specific transfer routines for a run. Because intermixing of floating and fixed point computations within the same program is rarely encountered, DAC and ADC lists have been restricted to include either floating point variables or fixed point variables, but not both. This enables performance of ADC/DAC functions in simple indexed loops which are easily tailored.

## Dynamic dump

The capability for dynamically dumping variable values onto bulk storage during a run and processing them later when time becomes less restrictive, is a desirable feature in any hybrid system. In addition to providing information for analysis purposes, it is very useful for dynamic debugging. Two essentially distinct functions are associated with a dynamic dump capability. The first involves the specification of those variables which are to be dumped and the actual performance of the dump during execution. The second involves the capability to display either the same variables that are dumped or a set of variables which are derived from the original variables by a user written processing program. Three user requirements affect the specification of the dump function. First, he must have freedom to specify those variables which he wishes to dump and the frequency at which they are to be saved. Second, for ease in interpreting the results, the values dumped should be coherent in time. That is, all values saved from a given interrupt level should represent functions of the same time, otherwise, a time skew in interpretation of the results will occur. Third, if post run processing of data is present, the user must be allowed to easily specify the form of process and a display list that is different from the list of variables dumped.

### Mode control and interrupt processing

In considering the most suitable form for mode control and interrupt handling routines, the situation is somewhat different from that of inter-console data transfer. Usually the programs necessary to handle these functions are very hardware dependent and generally so complex that only a highly experienced programmer can adequately cope with the problem involved. Here the obvious approach to specifying executive requirements is to remove flexibility, and therefore, the need for user intervention from the system. Some user control is necessary, however, and the amount of flexibility allowed by the executive should be sufficient to satisfy his reasonable needs. Certainly the user must be permitted to specify what subprogram he wants executed when the computer is in a given mode or when a particular interrupt occurs. He must also be able to specify the priority of each interrupt. It is also reasonable that an executive should expect the user to specify the frequency and perhaps the source of an interrupt. Beyond these few items, it should not be necessary and, in fact, it is not desirable for the user to intervene in the operation of mode or interrupt control software. The other user consideration that should be included is a "no penalty clause". Thus, if a user requires only three interrupt levels, he should not be required to inform the system that the other available levels are not required. In general, the user should only be required to specify those items which he needs for solving his problem.

The procedure required to specify the specific interrupt structure for a given problem is as follows. The address of a list is passed as a parameter to a standard executive routine which tailors a general interrupt structure to meet the users requirements. The list includes the names of the subprograms included in the real time program and the names of their associated data blocks. A similar list method is used to specify routines that are to be executed when the computer is placed in a given mode.

The standard executive routine is written such that it completely handles all normal mode control and interrupt servicing. Dummy subroutine calls are included to allow user definition of special mode or interrupt routines. During initialization the executive extracts information from the lists described above and modifies the dummy calls with appropriate user supplied routine addresses. Similar dummy instructions are used to permit generalization of other functions. Since all dummy entries initially consist of "NOP" instructions, failure to specify all modes or interrupt levels will not affect execution.

It was claimed earlier that the structures described serve to simplify the preparation of individual subprograms and the correlation of these into a unified system. A review of the necessary steps will illustrate this. While writing a subprogram, the user must only be aware of the name assigned to the subprogram, the name of the data block associated with it, and the names of the executive sub-routines he wishes to call. The total number of names needed in the TRW executive is six.

Integration of the subprograms demands very little more from the user. Before final compilation of the real time program, lists defining the mode control and interrupt structure must be prepared. Since this is done very late in the development of the program, all information is readily obtainable. Preparation of lists describing the details of interrupt priorities, execution intervals, etc., may be left until computer runs are planned. Since the entire problem should be well defined at this time, little difficulty is encountered in selecting specific values for these parameters.

## Static check

A major task which must be performed in any simulation is static verification of both the hardware used and the program being executed. An effective digital program can greatly aid in carrying out many parts of this task. The items that can be provided by the digital computer system for static checking are:

- Initialization of the system using parameter values chosen for the check.
- Comparison of computer values determine from the physical equations in the digital with that those values sampled from the analog.
- Information useful in verifying the validity of the equation values computed by the digital, is, debugging aids.

At TRW, the first requirement is met by the normal executive system used for analysis runs. When a static check is requested, normal run setup procedure is followed to the end of the initialization phase of the real time program (Figure 2). At this point, the static test request is recognized and execution of the static check program begins. Using this method of establishing the check case provides the advantages of convenience and flexibility in three ways. First, it allows rapid switching to the check mode using actual run values if a problem arises during analysis. Second, after the check is made and the problem corrected, the return to normal running conditions requires absolutely no action. Third, the system allows rapid definition and execution of several different check cases. All that is necessary to perform a static check is the entry of desired parameter values, using exactly the same methods as any other analysis run, and a request for execution of a static check. Since defining a single check case that effectively verifies an entire analog program is virtually impossible, the ability to perform a series of checks is very important. Proceeding through the initialization phase of the real time program has the advantage that the ADC and DAC values which are sampled and presented during initialization represent realistic problem values. This is sufficient to complete the set of values needed to base the entire static check on direct evaluation of the physical equations.

The static check overlay consists of two programs. The first is a FORTRAN subroutine in which the user codes his equations for use as the check reference. Because the system is dependent upon having access to normal run parameters which are stored in the computer COMMON area, the use of FORTRAN was a natural choice. Also, the use of FORTRAN rather than an interpreter program does not constrain the user in coding the analog equations, as encountered in some executive approaches.

The second program comprises the executive part of the overlay (Figure 4). It compares the equation values computed in the user FORTRAN program with the output of an analog component and generates appropriate error messages. The address of the analog component, the analog scale factor, and two indices which are used to correlate the component and the appropriate equation value are entered as a data record. Since only physical equations are coded in the FORTRAN program, recompilation is necessary only if these equations are redefined.

The correlation of an equation to an analog component and scale factor is achieved by using two indices specified in the FORTRAN routine as follows. The terms or factors of an equation that appear at a particular analog output are coded individually and stored in a one dimensional array. The section of coding for each equation is identified by a statement number or index. The statement number and array index are then included on a data card with the component address and scale factor to provide the necessary correlation. Since the computation of the terms of an equation is done only after a complete set of data cards for a given equation is read, the array used need only be large enough to store all of the values computed for the largest equation.

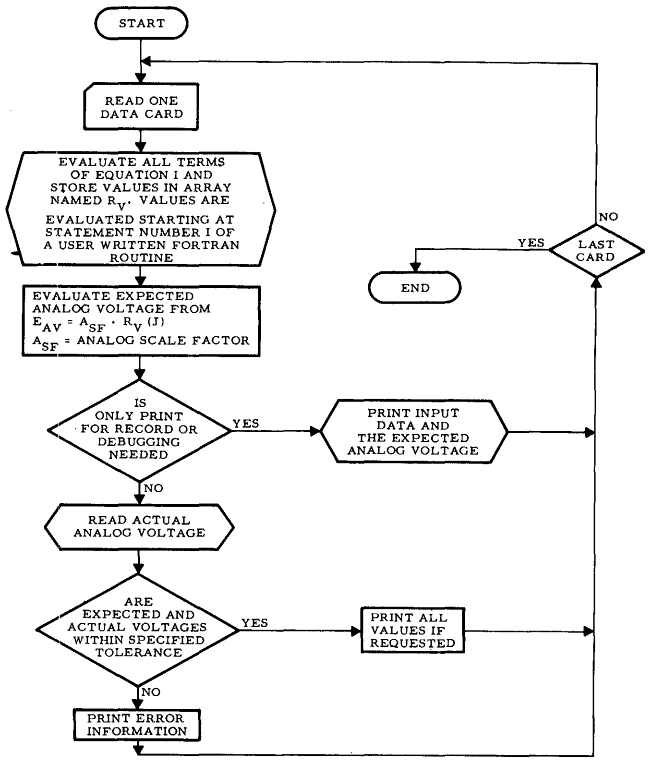

START
READ ONE DATA CARD
EVALUATE ALL TERMS OF EQUATION I AND STORE VALUES IN ARRAY NAMED $R_V$. VALUES ARE EVALUATED STARTING AT STATEMENT NUMBER I OF A USER WRITTEN FORTRAN ROUTINE
EVALUATE EXPECTED ANALOG VOLTAGE FROM $E_{AV} = A_{SF} \cdot R_V(J)$ $A_{SF}$ = ANALOG SCALE FACTOR
IS ONLY PRINT FOR RECORD OR DEBUGGING NEEDED
YES
NO
PRINT INPUT DATA AND THE EXPECTED ANALOG VOLTAGE
READ ACTUAL ANALOG VOLTAGE
ARE EXPECTED AND ACTUAL VOLTAGES WITHIN SPECIFIED TOLERANCE
YES
NO
PRINT ALL VALUES IF REQUESTED
PRINT ERROR INFORMATION
LAST CARD
YES
NO
END

Figure 4—Static check control

The executive also provides user options that allow extensive verification of the user program and the data files without requiring the presence of an analog computer. This option is usually not available in interpreter programs. The first option is a data card editing function that detects obvious format and keypunching errors. The second performs the normal static check procedure but replaces the interrogation of the analog computer with a printout of both the actual and scaled equation values. This data may then be used to do off-line debugging of the static check routine. Use of this feature can assure that only analog program debugging will be necessary when the analog computer is finally checked.

Other options are available to provide flexibility. One allows a choice between checking all analog components or just components that represent the total value of an equation. In the latter case, an error in the final equation value will direct the program to check all of the terms of that equation. A second option permits skipping a check of selected parts of the program.

## Operating procedures

Operationally the TRW executive has proven very effective. The entire procedure for executing a computer run consists of entering desired parameter values and a single command "RUN". From that point, all setup, operating, and display functions are performed automatically in a manner predefined by simple list inputs. The provision that analog setup routines have access to normal data parameters is of course the key to making such a simple run procedure possible.

### *Future hybrid executive development*

As it was indicated earlier, the TRW executive was developed through a process of evolution under the pressure of developing concurrently a large simulation. Although the operational characteristics which have resulted from this evolution are generally very good, many of the systems software aspects leave room for development. With a recent expansion in the number of systems software personnel at TRW, it is now possible to reimplement the executive on a sounder systems basis and integrate it into a more comprehensive software system. As proposed, the new system will provide a multi–user capability, simplified file processing, a more powerful I/O structure, accounting control and extensive debugging aids.

## ACKNOWLEDGMENT

The authors wish to express a special appreciation to Charles E. Vaughnn for his contributions in the design and especially in the implementation. It was by his outstanding efforts and his expertise on the CDC 3100 digital computer that the details of the design were worked out to assure compatibility throughout the executive and with the system software.

## BIBLIOGRAPHY

1 G A BEKEY
*Hybrid computation*
John Wiley & Sons Inc N Y 1968 7 177

2 D R MILLER G N GRADO B R BAKER
*The philosophy and the result: Comcor's CI-5000 hybrid computing system*
Simulation July 1965 39-46

3 T D TRUITT
*A discussion of the EAI approazh to hybrid computation*
Simulation Oct 1965 248-257

4 B R WILSON
*The Boeing integrated hybrid operating system*
Simulation Nov 1967 209-223

5 R B McGHEE A Y LEW
*Software for hybrid computers*
Simulation Dec 1965 367-373

6 C K BEDIENT L L DIKE
*The Lockheed hybrid system - A giant step*
Proc FJCC Vol 33 Part 1 1968

7 G N SOMA J D CRUMKLETON
*A priority interrupt oriented hybrid executive*
Proc FJCC Vol 33 Part 1 1968

8 M D THOMPSON
*Growing pains in the evolution of hybrid executives*
Proc FJCC Vol 33 Part 1 1968

9 D A WILLARD
*The Boeing/Vertol hybrid executive system*
Proc FJCC Vol 33 Part 1 1968

10 E A JACOBY J S RABY D E ROBINSON
*Family I: Software for NASA-Ames simulation system*
Proc FJCC Vol 33 Part I 1968

11 W GILOI D BECKERT H C LIEBIG
*A flexible standard programming system for hybrid computation*
Proc SJCC Vol 34 1969

# A system for clinical data management

*by* R. A. GREENES, A. N. PAPPALARDO, C. W. MARBLE, and G. O. BARNETT

*Massachusetts General Hospital*
Boston, Massachusetts

## INTRODUCTION

The application of computers to the delivery of patient care is more a problem of "data management" than of "data processing." Although calculations and interpretation of data are often required, of much greater concern are the problems involved in the collection, communication, coordination, and presentation of information. As the process of delivery of medical care becomes increasingly complex, and involves increasing numbers of professional and nonprofessional personnel, responsibility for achieving the continuity and comprehensiveness that is essential to medical care seems to rest heavily on the development of appropriate computer-based data management systems. Such systems may further provide the primary feasible means by which quality control, auditing of the medical care process, and research into the diagnosis and treatment of disease can be achieved.

These functions now are dependent on the use of the patient medical record, although they are fulfilled only to a minimal extent by it. Despite changing functions and increased demands on it, the medical record has changed little in form over the past century. Medical records possess no organization by diagnostic or therapeutic problem; notes relevant to a particular aspect of a patient's health may be accessed only by leafing through an entire volume. Terminology is not standard, data is not organized in well-defined formats, and notes are often illegible. As a consequence, the objective of using the computer for clinical data management is gaining considerable impetus.

This paper will describe a number of criteria which the authors have found to be important in the design of systems for clinical data management, and a novel system which has been implemented to meet these requirements. The system to be described has been in operation for over a year. The extent to which it has proved useful has led the authors to believe that the criteria defined have general applicability for clinical data management. In the discussion to follow, the term "clinical data management system" refers to a time-shared computer system which supports on-line input, inquiry, and retrieval of clinical information from a central data base.

### *Design and implementation*

The internal design of an information system dictates constraints on the external attributes of such a system. The characteristics that must be resolved include the number, priority, and level of responsiveness of the users, both active and inactive; the ratios among CPU time, connect time, and input/output time; the structure, magnitude, and timeliness of file information; the profile of application programs in regard to size, type, and interactiveness; user requirements for development and service modes of operation; and finally, the overall economic justification for the system.

## High level programming language

One of the most time-consuming aspects of the development of information system programs involves the optimal interfacing of the system with its users in a particular application area. This requires much attention to human engineering, and repeated modification and revision of programs. The implementation of clinical data management applications has generally begun on relatively small computers. This has, in many cases, been necessary because development was a gradual process and started with limited objectives. Since high level languages have not typically been available on small machines, most programming has been done in machine language.

The expense and inefficiency of writing, debugging, and modifying such programs have been serious obstacles to active research and development. A few clinical data management systems have used large general purpose computers which could provide much increased flexibility. However, the overhead of a large operating system on a major computer has often seemed excessive, because of the rather small amount of processing involved in many of these applications. Futhermore, because of the reliability requirements of a clinical data management system, modularity and duplication of hardware is desirable and often essential. Because of the expense entailed by hardware redundancy, this is typically feasible only with inexpensive, minimal equipment configurations.

The MGH Utility Multi-Programming System-(MUMPS) is a compact time-sharing system on a medium scale computer, dedicated to clinical data management applications. It is currently implemented on a PDP-9 (Digital Equipment Corporation) with 24,000 words of 18 bit memory and a Burroughs fixed head disk with three million characters of storage capacity. A set of terminal scanners is used to interface to remote devices: teletypes, buffered display scopes, line printers, card readers, and A/D converters. Both memory size and peripheral storage capacity can be expanded in the system. In the current version, 16 users may run simultaneously.

All application programs in this system are written in a high-level interpretive language, a distant ancestor of which is JOSS,[1] developed at the Rand Corporation in 1964. It has also been influenced by related languages such as STRINGCOMP (developed by Bolt, Beranek and Newman, Inc.), and FILECOMP (specified by Medinet Division of General Electric Corp.). The MUMPS language allows the programmer to write a program, debug it, edit it, run it, and modify it concurrently during an interactive session at a console. The interpreter itself is a part of the executive system and is re-entrant. The total space taken up by the time-sharing monitor, the I/O monitor, buffers, and re-entrant interpreter is currently about 8,000 words of memory. The time-sharing and I/O monitors have been specifically tailored to work efficiently with the interpreter. No attempt has been made to accommodate

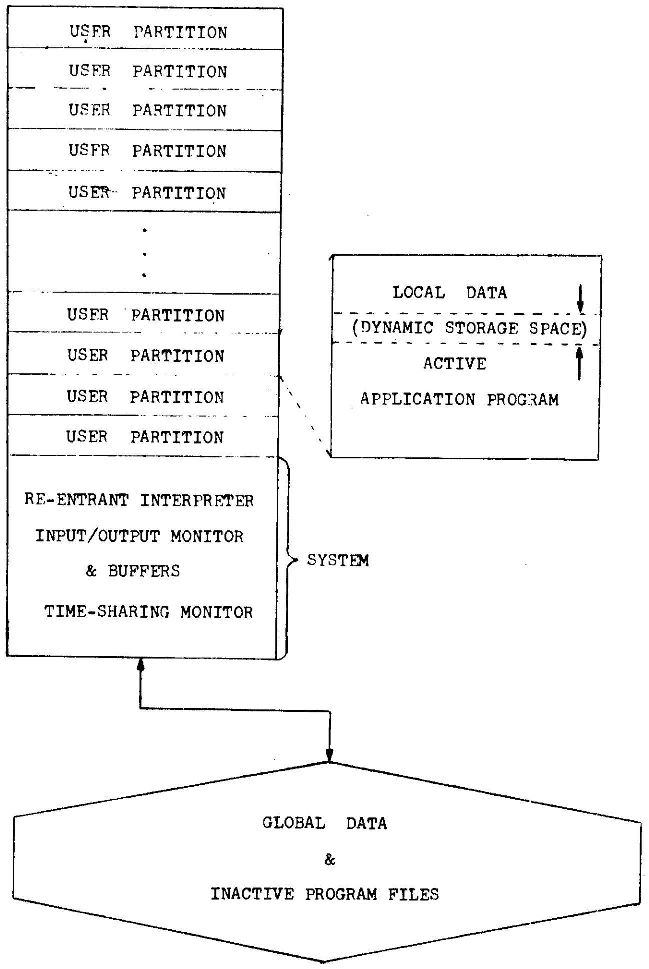


Figure 1—A schematic diagram of the core memory allocation of the MUMPS system and user partitions. A single partition is expanded to show its internal structure. The use of secondary storage (disk) for global data and inactive programs is represented.

machine language user programs. All active users are assigned partitions of core memory. Activating a program consists of finding an available partition and bringing the program into it from disk; as long as it remains active, it occupies its partition. Core and disk storage allocation are depicted in Figure 1.

The basic orientation of the language is procedural, much as FORTRAN and ALGOL. The largest unit of a program is a group of statements called a "part" indicated by an integer part number. A single line or statement of the program is a "step"; it is identified by a step number consisting of a decimal fraction appended to the part number. Multiple commands may be entered in a single step and executed one after another. A conditional statement which when evaluated has a false value will, however, cause the rest of the commands in that step to be ignored. Commands may be stated in a long mnemonic form, or for the experienced user, in a much more compact form in which only the first letter of the command is used. A statement preceded by a step number is considered to be in "indirect" or "program" mode, and is stored to be executed as part of a program. A statement without a step number is in "direct" mode, which indicates that it is to be executed immediately after it is entered from the user terminal.

### Interface flexibility

Clinical information about a patient derives from a variety of sources—the patient, the attending physician, consultants, the radiologists, the clinical laboratory, etc. Problems of using the computer to obtain information from each of these sources have begun to receive attention. Perhaps the most widespread activity of this type has been the development of systems for clinical laboratory information processing.[2,3,4,5]

With the exception of laboratory data, which is either numeric or simple text, much of the clinical information in the medical record is generally recorded in narrative or free text form. Most investigators are convinced that natural language is not in general suitable for computer record keeping applications, except perhaps in certain circumscribed areas with limited vocabulary and syntax.[6,7] As a result, there is a significant amount of work currently being devoted to the development of methods for structuring this narrative data.[8,9,10] It is generally recognized that this may be best achieved by introduction of new ways of capturing such information, e.g., entry of data by use of check lists, forms, or direct user-computer dialogue. Interactive dialogues for the capture of narrative data may be based on hierarchical organization and presentation to the user of the subject material. Any particular topic may then be pursued to an arbitrary depth, by means of a succession of increasingly discriminating selections by the user from the options presented. A variety of programs for interactive acquisition of clinical data have been developed, and have generated needs for special terminals, display formats, and conversational languages. Conversational programs have, for example, been devised for the on-line acquisition of a patient's medical history.[11,12] Other systems aimed primarily at the physician have been designed for the purpose of entry of physical examination notes,[13] the recording of progress notes, or the generation of X-ray reports by the radiologist.[14,15] In the development of such applications, the emphasis is placed primarily on the interface (hardware, software, and environmental) of the system with the individuals who have to use it.

As the potential of clinical data management systems is recognized, they will be called upon to fulfill a diversity of output functions, e.g., the display of reports or summaries, organized chronologically or topically, the production of tables or graphs. Information obtained by dialogue must often be translated into more precise medical terminology, or compacted into coded representations. Flexibility in output and presentation of information, as well as in its acquisition, is essential.

The philosophy of MUMPS has emphasized the need for ease in interfacing and adapting programs to the requirements of the application. Programs written in the interpretive language do not require any compiling or assembling. Error comments during execution are typed out at the user's console, and allow quick recovery, modification of the program, and reexecution of it. All debugging and modification is done in the same language in which the program is written and can be done entirely from the user terminal. This makes modification especially convenient, particularly in a service environment where the trouble shooting necessary to interface a program with an application area is a time consuming process. The MUMPS environment allows a programming session to take the form of a conversational dialogue between the programmer and the terminal device, thus minimizing the user's time in programming a problem, the computer's time needed in checking it out, and most important, the elapsed time required to obtain a final running application program.

### Text handling capabilities

The complexity and variety of data that must be handled in a clinical information system impose a number of requirements on the system. A considerable amount of information that is input is in the form of text strings of variable length. The processing of input often requires syntax checking or limit checking. String comparisons, extractions, and concatenations need to be performed. When special driver languages or monitor subsystems are employed to control dialogues between the user and the computer, string processing capabilities are mandatory. Most existing higher level languages do not provide the needed combination of algebraic and boolean expression handling capabilities with the ability to handle string information.

The MUMPS language has been designed to meet this need. In addition to algebraic and boolean processing, a MUMPS program can perform string extraction, locations, comparisons, and checking of

```
→WRITE 1

1.10 READ !,"UNIT NO.  ",X
1.15 IF 'X:3N"-"2N"-"2N TYPE "  ILLEGAL" GOTO 1

→DO 1

UNIT NO.  123-45-678  ILLEGAL
UNIT NO.  12-345-67  ILLEGAL
UNIT NO.  123-456-78  ILLEGAL
UNIT NO.  123-45-67
```

Figure 2—A portion of a MUMPS program to input a seven digit unit number from the teletype (accomplished by step 1.10. The value entered is stored as the variable named X; a check is made that X has the correct form, i.e., 3 digits, followed by a hyphen, 2 digits, a hyphen, and 2 more digits (step 1.15). Improper values cause an error message, and request of a new value. The WRITE command lists the statements. The DO command causes execution, which is illustrated. (In this and other figures, user input is underlined to distinguish it from the response of the computer.)

syntax and form of information. These features are illustrated in Figures 2 and 3. Figure 2 shows a portion of a program written in MUMPS to read a hospital unit number from a Teletype (i.e., entered by a user), to check its syntax, and to reject any improperly formatted responses. Figure 3 shows statements in a program for the clinical chemistry laboratory, which permit entry of a test name and its result. Checks are made on the legality of the test name and the reasonableness of the result. Some of the interactive editing capabilities are shown in the figure.

## Terminal device flexibility

An important feature of the language is its input/output scheme, which permits programs to be written independently of the particular device for which one is programming. One may use any device for which the hardware system has been appropriately interfaced by merely assigning a device number to a system variable indicating the device to be utilized. This makes it possible to generate a report on a display scope, for example, and then to use the same program to type out the report on a typewriter, merely by changing, during execution, the value of the device number assigned to the input/output variable. Formatting and control of position on a page are made very simple by utilization of special format characters and variables indicating current position and line spacing.

## Multi-user access to a central data base

A major requirement of a clinical data management system is that the information stored be accessible to a variety of users concurrently. Access may be from a variety of terminals, by a variety of programs in the system, at varying frequencies. Among the possible purposes for accessing a file might be to report a laboratory result, to enter an X-ray impression, to record a progress note, or to enter a specific inquiry. Although many of these activities occur independently, they must share a common data base. Nevertheless, manipulation of the data base must occur without time sharing conflict, such as might occur if two users were to update a portion of the data base simulta-

```
→WRITE 2,9

2.05 SET DCT="CA,P,FBS,CHOL,TP,NA,K,CL,CO2,SGOT,LDH,VDH,BUN,CRE"
2.10 READ !,"TEST: ",TES
2.20 FOR I=1:1:14 IF $PIECE(DCT,I)=TES QUIT GOTO 2.3
2.25 TYPE " ???" GOTO 2.1
2.30 ASK !,"RESULT= ",RES GOTO I+3
2.40 READ "  PROB. ERROR...OK? ",X IF 'X["Y" GOTO 2.3
2.50 DO 100 TYPE ! GOTO 2.1

9.10 IF RES>160!RES<120 GOTO 2.4
9.20 GOTO 2.5

→DO 2

TEST: MA ???
TEST: NA
RESULT= 125

TEST:
? 2.10 IOINT

→9.1 IF RES>150!RES<130 GOTO 2.4
→DO 2

TEST: NA
RESULT= 125  PROB. ERROR...OK? Y

TEST:
```

Figure 3—A section of a MUMPS program that might be used in a clinical chemistry laboratory information system. Step 2.05 sets the variable DCT to the list of test determinations that are valid for this particular laboratory. Step 2.10 then accepts a test name from a technician. The $PIECE function in step 2.20 then extracts substrings (between commas) from DCT and compares them to the variable TES whose value is the test name entered. It does this repeatedly for values of I = 1,. . . , 14 until a match is found; at this point the iteration is terminated and execution continues at step 2.30. If no match is found, an error comment is printed (step 2.25) and step 2.10 is repeated. Step 2.30 accepts a test result, and goes to a part in the program dependent on the particular value of I for which the match was found.
Part 9 illustrates a specific check for results entered for the test name, NA (in which case I = 6). The result is compared to prescribed limits, in step 9.10, and if it exceeds either limit, control goes to step 2.40. Here the user is asked to verify the value. The user's response is inspected to see if it contains a "Y", in which case a YES response is implied. Otherwise, a new result is requested, in step 2.30. If either the user verifies it, or the result is within limits set by step 9.10, control goes to step 2.50. Step 2.50 calls part 100 to file the value and then returns to step 2.10.
The DO command causes execution, which illustrates operation of the program. Note that the user has interrupted the program from his teletype (indicated by the "? 2.10 IOINT" error comment, showing where the interrupt occurred). In this case, a programmer has decided to edit the program to make the limits for a sodium determination more stringent, by retyping step 9.10. The program is then re-executed.

neously. Without special provision, this migh tresult in loss of information.

Efforts to develop specialized clinical data management applications are still relatively primitive. There have been very few concerted efforts devoted to the general problem of management of medical record data, the development of integrated patient data files, and the implementation of systems for long term storage and retrieval of this data.[16,17] Among the difficulties faced by the few developmental efforts that have been undertaken have been the lack of generality in their approaches, and the reliance on highly specific programming languages, file structures, and file handling routines.

MUMPS provides application programs with the ability to create and utilize their own "local" data, as well as to manipulate "global" data, shared by other programs in the system. Local data utilized by a program is referenced symbolically, and space for it is allocated as needed. Local data is that set of variables established within the domain of a particular program, and available and defined only within that program. The data actually resides within the user partition, and functions as scratch or transient data. Local arrays are assumed to be sparse or of varying dimensions, and only subscripts for which data are defined are allocated space. A symbolic variable used in a program may be given either a numerical value or a variable-length string value. When it has a string value, only that space required by the string is actually allocated. Thus for both strings and sparse arrays, the overhead of a compiler system does not exist, in which typically maximum sizes of arrays and maximum lengths for string variables must be allocated.

This philosophy is extended to the management of data on the random access disk. Elements stored in data files are referenced entirely symbolically; the file name is similar to that of a local variable name in a program. Fields in the data file are treated as array elements and referenced by means of subscripts; subfields are referenced by appending additional subscripts. Data files on the disk thus comprise an external system of arrays, which provide a common data base available to all programs. The arrays which make up this external system are called global variables, and are identified by global array names. A global name (or file name) consists of the character up-arrow (↑) followed by at least one alphabetic character. The form of the subscript portion of an array reference consists of an arbitrary number of numeric expressions separated by commas and enclosed by parentheses.

To avoid time-sharing conflicts, a program may prevent other programs from having access to one or more global arrays which it is in the process of altering in some way, by the use of the command OPEN. The argument of OPEN may be one array name or a list of array names. OPEN prevents any other program from altering data in any of the specified arrays. The effect of OPEN is cancelled when the program ends or at the occurrence of the command CLOSE, which does not require any arguments, and releases all opened arrays to other users in the system.

## Hierarchical data base organization

A most important requirement for clinical data management is the ability to handle the several levels of structure of a medical record data base, and to support the rather complex updating and retrieval needs of such a system. An example of a typical patient data file, such as exists in the information system under development at the Massachusetts General Hospital, is illustrated in Figure 4. This indicates the typically hierarchical (tree-like) structure of the data base, which has both a topical and a chronological organization. Most computer systems currently available do not have the ability to utilize hierarchical file organizations conveniently.

The global array facility in MUMPS has been designed to meet this need. The structure of global arrays is hierarchical, and any node within the array tree may possess a numeric or string data value and/or a pointer to a lower level in the tree. Data may be stored at any level, and there are no constraints to the dimension or the size of the array. In addition the quantity and magnitude of subscripts for an array are dynamic, so that not only may the content of an array change during usage, but also its structure may vary.

Since modification of content and structure of a global array may be caused by a variety of programs in the system, a particular program must sometimes examine the current configuration of an array before attempting to access or update it. MUMPS provides a set of global array functions to determine the type and structure of a global array. These functions permit the programmer to locate the nodes where information is stored within an array, and nodes within the array which are empty and thus available for data storage.

The storage of data into an array is accomplished solely by the assignment command, SET. Consider the following statement:

```
SET ↑APR(UN,NAME)="JOHN DOE",
↑APR(UN,AGE)=34
```

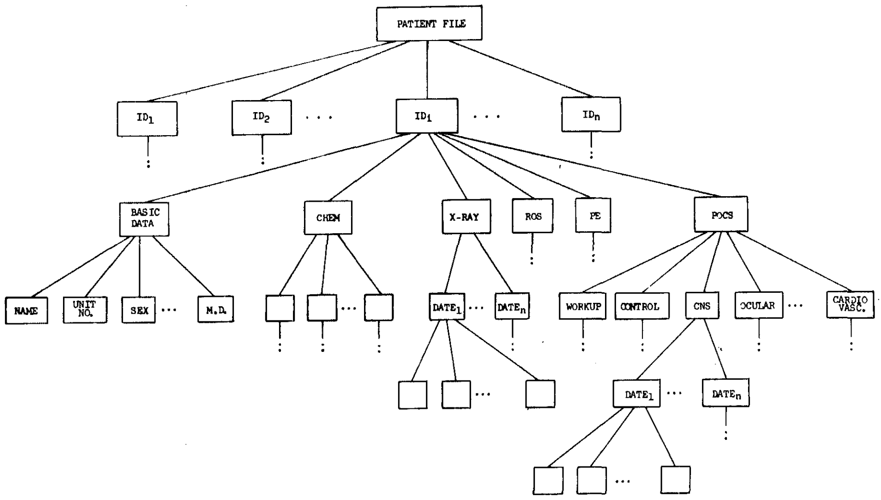


Figure 4—A tree-structured patient data file, indicating: (1) the use of certain levels in the tree to group information in specific topics, e.g., basic identifying and administrative data, review of systems, physical examination, and (2) other levels to group information into sets which differ by date or by some other sequencing field.

Assume the global array name ↑APR is reserved for the active patient record file. Each patient in the file is accessed through his hospital unit number, in this case, a local variable UN. Both NAME and AGE are also local variables whose values indicate particular categories represented by subscripts at the second level of the array. This statement then assigns the string value "JOHN DOE" and the numeric value 34 to the specific second level categories, name and age respectively. Subsequently, a statement such as:

```
SET ↑APR(UN,CHEM,N)=DATE.",".TEST
```

might define the Nth laboratory test in the chemistry lab with the double field entry of the date concatenated (by means of the dot operator) with a comma and the test name.

Retrieving data from global arrays is no different from retrieving data from local arrays. Both consist of ascertaining the value of a subscripted variable by using it within a numeric or string valued expression. The statement:

```
TYPE " THE AGE OF ", ↑APR(UN,NAME),
" IS ", ↑APR(UN,AGE)
```

will effect the printout:

```
THE AGE OF JOHN DOE IS 34
```

To print out a list of a patient's laboratory tests (assuming ↑APR(UN,CHEM) is the total number of tests defined) the following statement might be used:

```
FOR I=1:1:↑APR(UN,CHEM) TYPE
↑APR(UN,CHEM,I)
```

The KILL command when applied to a specific node in a global array, prunes the array tree at that node. Any data value and/or array pointers to lower level nodes are removed, and that node reverts back to an undefined status. The statement KILL↑APR (UN) would delete all information for the patient defined by the local variable UN.

Included in the global array syntax is the "naked" global variable. The form of the naked variable consists of the up-arrow followed by a subscript enclosed in parentheses. This notation is equivalent to the last previously used global array reference except that the value of the last subscript is replaced with the value of the subscript in the naked variable. For example, the statement:

```
TYPE " THE AGE OF ",↑APR(UN,NAME),
" IS ",↑(AGE)
```

is equivalent to the example cited earlier.

MUMPS requires that reference to all file information be done symbolically, in the syntax of hierarchical global arrays. This replaces the classical manner of sequentially accessing record files on secondary memory devices. Instead, an attempt is made to logically map the content and structure of the tree-like data arrays into the physical storage medium of the system. The general technique is to map logical information at a specific level of an array into fixed size blocks chained together linearly to contain all the data values stored at that level, and all the pointer words which link it to the chains of the next lower level. The implementation of this design requires a careful consideration of the timing and size constraints of the physical device in relation to the overall system. The actual memory device used in the system is a large fixed head disk. The organization of this type of disk is two dimensional, wherein any physical block has a track and a segment coordinate. Initially a set of free lists are formed which chain all blocks possessing the same segment address together. Whenever a continuation block at the same level or a header block at a new level is required, the appropriate block in the free list whose segment address is a few segments away is utilized. This method makes it possible to trace down the many levels of a tree structure required to access a datum during a fraction of a disk revolution, in addition to the average access time of the disk unit required to reach the first level of the tree. As a consequence, the time required to retrieve a particular datum is virtually independent of the depth of subscripting required to specify the datum. Space is conserved by utilizing small sized physical blocks such that at any subscript level an average of one continuation block is required. When data is updated, care is given to repack and sometimes reorganize the individual data elements within a chain to insure maximum utilization of space for variable length data. Whenever a part of the global structure is deleted, it is passed to the garbage collector routines to be disassembled from tree-like chains back into linear chains and appended to the appropriate free lists. This is done during periods of low CPU activity so as to avoid competition with the active programs.

Once a block of data accommodating a single level of subscripting is referenced, it is maintained in core memory until a reference is given to a different block by the program. Use of the naked variable then permits other data at the same level to be referenced merely by specifying a terminal subscript, so that once a level is reached, often no further disk access need be made to manipulate associated information. If any data in a block is altered, it is only written back on the disk when a reference is made to a block other than the one that is in core memory, or when a CLOSE command is given.

### Large storage capacity

The conversational environment in which a clinical data management system is designed to operate demands little computer processing power. When data is entered, a program need only check on its legality, decide where to file it, and select an appropriate response to the user. Generation of reports may involve manipulation of information from peripheral storage to assemble the data needed, but only a small amount of processing to actually format or produce the report. Large volumes of data need to be available for low level, low frequency usage. Thus one does not need computing power as much as the availability of peripheral storage of large capacity. Much of the data may be potentially accessed at any time, and therefore need to be stored on a random access device. Because of the large quantities of data that may be anticipated in such systems, it is necessary to provide hierarchies of peripheral storage, in which the access time of the storage device used is commensurate with the frequency or urgency of the need for retrieval.

In MUMPS the fixed head disk provides fast random access storage, whereas slower access requirements are currently met by three Dectape units. A large movable head disk unit is being installed to permit intermediate access times for other data.

### Efficient Time Sharing

In a conversational data management system, programs spend much of their time in an input/output hung status, i.e., doing disk activity or completing a transaction at a terminal. As a result, there is again not a large demand by a program for the central processor. In contrast to most numerical applications where central processing power is the limiting factor, in a conversational environment the time necessary to complete a task is often determined by the speed of the input/output equipment or the human response time at a terminal. As a consequence of the small demand for the central processor by an individual program, one can theoretically time share a large number of programs. Efficiency of the use of the central processor is in this situation determined by how rapidly the time-sharing monitor can change from one user to another. This swapping overhead is the delay before a particular user program can run after a previous user has quit the run state, due to an input/output hang, expiration of time slice, or termination of its task. When the central processor is not being fully utilized, swapping overhead tends to determine response time of the system.

TABLE I—A comparison of execution times for various numeric processing examples in MUMPS and FORTRAN

| Statement | CPU Time (Microseconds) MUMPS | FORTRAN | MUMPS/FORTRAN RATIO |
|---|---|---|---|
| FOR/DO (Iteration, per cycle) | 250 | 12* | 20.8 |
| 1 + 2 | 800 | 7* | 114.3 |
| 2*3 | 850 | 44 | 19.3 |
| 1 + 2*3 | 1050 | 48 | 21.9 |
| 1 + 2 − 3*4/5 | 1550 | 120 | 12.9 |

* These are the only operations compiled by the PDP–9 FORTRAN Compiler as in-line code. All other operations beside integer addition (in DO loops and arithmetic expressions) are compiled as subroutine calls.

In the MUMPS system, the use of a partitioned memory has been dictated by the overwhelming concern for response time. As a result of partitioning, the time sharing monitor can switch between users in minimum time without having to resort to swapping of programs in from a drum or disk. In addition, the monitor automatically overlays external program segments invoked by an active program. Proper linkages are set up to return automatically to the invoking program when execution of a segment terminates.

Execution speed of an interpretive program doing pure numneric processing may be slower by a factor of about 20 to 1 over corresponding code generated in a compiler or assembly language system.

Table I illustrates some timing comparisons between a single user version of the MUMPS interpreter and the manufacturer-supplied FORTRAN compiler for this computer, for statements involving pure numeric processing activity of varying complexity. As has been indicated above, however, few programs do pure numeric processing in a clinical data management environment. Input/output conversion in FORTRAN and most other compiler systems is handled in a purely interpretive fashion, and thus, for this activity, very little difference in the performance between the two kinds of systems may be expected. Furthermore, a significant part of the processing done by programs in clinical data management systems involves file manipulation, or text string processing activities; in all assembly or compiler language systems these functions are usually handled by the use of subroutines. Therefore, the employment of an interpreter as a means of generating calls to these subroutines rather than compiling the calls themselves requires only a small amount of processing overhead.

The foregoing observations refer to comparisons between execution speeds of MUMPS interpretive language statements and compiler-generated object code on a single-user computer, with no other processes competing for the processor. More significantly, in a data management environment, a re-entrant interpreter such as MUMPS may provide the most economical means of achieving a highly responsive time-shared information system. In the MUMPS system with sixteen typical users active, response times (a most sensitive measure of efficiency in a time-sharing system) are always less than a second and usually appear instantaneous.

There are several reasons that account for this, all of which are related to very efficient use of core storage. First, a typical program written in the interpretive language takes up 10 percent to 20 percent of the space taken up by the object code generated for a similar program written in a compiler language. Also, dynamic allocation of data and efficient storage of variable length strings and of sparse arrays are standard features of the interpreter. Thus data also take up considerably less space in this kind of environment. In addition, since the interpreter is re-entrant, all programs may share the same utility routines and operating system capabilities. This contrasts rather sharply with conventional compiler language operating systems, in which each running program must have its own copy of the necessary system routines that it will utilize.

The significant advantage that results from the above features is that programs take up much less space; therefore, a partitioned memory system on a medium or small scale computer becomes feasible. Active programs are typically highly interactive, and are therefore doing only small amounts of processing between input/output requests. Therefore the time-sharing monitor is invoked frequently to pass control from one user to another, in order to utilize the central processor as much as possible. In a partitioned system, swapping of the users is very rapid. In systems that use various schemes for submerging disk or drum swapping, users that are running in a conversational mode often do not stay in the run state long enough to submerge the concurrent swapping process. Therefore potential CPU time is unavailable; this unused time may be on the order of 20 to 50 percent of the total amount available. The speed that results from not using disk or drum swapping appears, in our experience, to more than offset the overhead of interpretation, with greatly increased efficiency in the utilization of space.

## CONCLUSION

The convenience occasioned by the utilization of a high level language with symbolic referencing capability for data stored in complex tree structures on peripheral storage has greatly simplified the development of application programs for clinical data management. This is the only system that we know of, on a computer of medium or small scale, which supports such extensive file manipulation, string handling, and input/output flexibility. It is the only system we have encountered on any computer which allows all these manipulations to occur entirely in a high level language. This system has been used at the MGH for all of our programming research and development activities. Equally important, because of its compactness and efficiency in this environment, we use it for the implementation of our service programs, including a chemistry laboratory reporting system,[18] a patient history taking system, and a number of programs for physician entry of narrative record information.

An advantage of this approach to clinical data management over the use of a large commercially available general purpose time-sharing computer-with its complex operating system has been the increased flexibility that is possible with a specially designed system. This increased flexibility results because the system has been built to meet specific objectives, in contrast to having been implemented within the often arbitrary and inefficient constraints of a general-purpose time-sharing facility. In addition, with a special purpose system, it is possible to achieve the efficiency required for service operation with a computer whose size and cost are well matched to the requirements of the problem area.

## BIBLIOGRAPHY

1 J C SHAW
*JOSS: A designer's view of an experimental on-line computing system*
Proc FJCC Vol 26 1964 455-464

2 D A LINDBERG
*Collection, evaluation, and transmission of hospital laboratory data*
Meth Inform Med July 1967 Vol 6 97-107

3 H C PRIBOR W R KIRKHAM R S HOYT
*Small computer does a big job in this hospital laboratory*
Mod Hosp Vol 110 April 1968 104-107

4 G O BARNETT P B HOFMANN
*Computer technology and patient care: Experiences of a hospital research effort*
Inquiry V 1968 51-17

5 G P HICKS M M GIESCHEN W V SLACK et al
*Routine use of a small digital computer in the clinical laboratory*
JAMA Vol 196 June 13 1966 973-978

6 H JACOBS
*A natural language information retrieval system*
Meth Inform Med Vol 7 Jan 1968 8-16

7 A W PRATT L B THOMAS
*An information processing system for pathology data*
Pathology Annual Vol 1 Century Appleton N Y 1966

8 G O BARNETT R A GREENES
*Interface aspects of a hospital information system*
Ann N.Y. Acad Sci (in press)

9 R D YODER
*Preparing medical record data for computer processing*
Hospitals Vol 40 Aug 16 1966 75-76

10 L L WEED
*Medical records that guide and teach*
New Eng J Med Vol 278 1968 652-657

11 W V SLACK G P HICKS C E REED et al
*A computer-based medical history system*
New Eng J Med Vol 274 Jan 27 1966 194-198

12 J G MAYNE W WEKSEL P N SHOLTZ
*Toward automating the medical history*
Mayo Clin Proc Vol 43 Jan 1968 1-25

13 J M KIELY J L JUERGENS B L HISEY P E WILLIAMS
*A computer-based medical record*
JAMA Vol 205 1968 571-576

14 A W TEMPLETON P L REICHERTZ E PAQUET J L LEHR G W LODWICK F I SCOTT
*RADIATE—Updated and redesigned for multiple cathode-ray tube terminals*
Radiol Vol 92 1969 30-36

15 H P PENDERGRASS R A GREENES G O BARNETT J W POITRAS C W MARBLE A N PAPPALARDO
*An on-line computer facility for systematized input of radiology reports*
Radiol Vol 92 1969 709-713

16 P HALL C MELLNER T DANIELSSON

# Medical education—A challenge for natural language analysis, artificial intelligence, and interactive graphics

*by* J. C. WEBER and W. D. HAGAMEN

*Cornell University Medical College*
New York, New York

## INTRODUCTION

In a functional sense, Computer Assisted Instruction (CAI) has not advanced from the primary grades, yet its implications for higher education cannot be ignored. Most of the work that has been done in CAI falls into the category of drill and practice or straight tutorial presentation. Logically, both the hardware and software that have been developed or modified to support CAI have been tailored with these goals in mind. In medical education, multiple choice questions would neither hold the interest of the average student nor challenge his intellectual abilities. Since we can formally present only a small fraction of the problems our students may some day have to deal with, we are concerned not only with presenting factual information, but even more with developing their power to reason and handle new problems. Medical students have widely divergent backgrounds and needs, as well as differing interests. For these and other reasons, we need a truly two-way, free-format discussion where each student is treated as an individual. Anatomy, the field in which we teach, is very much a visual science. Consequently, graphic capabilities are important. Here also the student needs to interact and be treated as an individual.

It should be pointed out that we are computer naive people who have been working without professional help. We have been using a system and a language which nicely meet the requirements for which they were designed, but in approaching the needs of higher education, programming becomes laborious and circuitous. We are well aware that others working with more sophisticated systems have produced more sophisticated results. Indeed, to many our methods may seem primitive. However, our challenge has been to implement natural language analysis, self-adaptive programming, and interactive graphics within a framework of restricted costs. It is important that people in the computer field be made aware of the systems and language requirements of people in various areas of education. For CAI ever to become a reality, it must first become an interdisciplinary endeavor.

### *The system and language*

Our work has been centered around the IBM 1500 Instructional System and its associated language—COURSEWRITER II. The 1500 is supported on 1130 (32 K) hardware. Peripheral equipment consists of 32 terminals, each with a cathode ray tube (CRT), a 128 character keyboard input and a light pen, a typewriter unit, and a 16 mm random access image projector.

COURSEWRITER II is an interpretive, non-computational language. Both COURSEWRITER II and the 1500 Instructional System are described in detail in IBM publications.

### *Natural language analysis*

Our basic format is schematically illustrated in Figure 1. There are two different types of discussions. The large circle represents an anatomical discussion

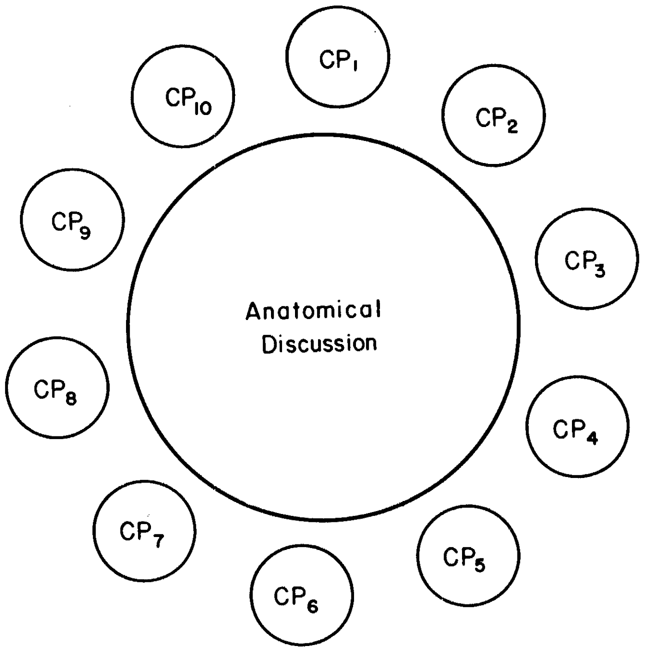


Figure 1—A schematic representation of the modular unit. The large circle represents an anatomical discussion, the smaller satellites represent clinical problems. A large number of these modular units are interconnected to form a course segment or topic of discussion

and the smaller satellites clustered around it represent what we call clinical problems. The cluster of clinical problems surrounding each anatomical unit is directly related to that block of anatomical material. We try to have a ratio of at least ten clinical problems to each anatomical discussion. Thus the organization is modular and any number of these modules may be linked together to form a course segment or topic of discussion. At the present time we try to keep these course segments small enough that the average student can complete them in 30–60 minutes.

To facilitate the description we shall consider a discussion of the extrinsic muscles of the eye and their nerve supply. There are seven such muscles and they are supplied by three nerves. This course segment consists of 13 modules, i.e., 13 anatomical discussions and their associated clinical problems.

A student signs on a terminal for a particular course segment, i.e., he chooses the general topic he wants to discuss. He is then presented with a choice:

DO YOU WANT TO BEGIN BY ASKING QUESTIONS? (SQ)

OR

DO YOU WANT ME TO INITIATE THE DISCUSSION? (CP)

If he indicates that he wants to ask a question, he is branched to a subroutine which handles the analysis of student questions (SQ). If he indicates he wants the computer to initiate the discussion, he is branched to one of the clinical problems (CP) in one of the clusters.

## Clinical problems

On the *initial* branch, i.e., if the student elects to have the computer initiate the discussion, both the cluster and the specific clinical problem in the cluster are randomly selected. Each clinical problem is a relatively brief linear presentation, i.e., three or four statements, each illustrated by a picture (with the film strip projector), followed by one key anatomical question. For example, after describing and illustrating a patient's signs, symptoms, and history, the student might be asked:

WHICH MUSCLE IS INVOLVED?

If he answered this question correctly, he would branch to another clinical problem in another cluster or module. The student is taken from cluster to cluster in a prescribed sequence. However, the specific clinical problem in each cluster is randomly, but non-repetitively selected. As long as he continues to respond appropriately, he branches from one clinical problem to another without ever entering into the underlying anatomical discussions. Each time he successfully completes a clinical problem, a scoring counter is incremented by one. If he were to progress through six of these clinical problems, he would have been examined on three of the seven muscles, and three of the seven branches of the three nerves supplying them. Since the general principles of function and methods of testing one muscle or nerve are similar to those underlying the others, it is our judgment that a student who successfully completes six *successive* clinical problems correctly, in this predetermined sequence, has demonstrated mastery of this block of subject matter, and he is told so. He may then either sign off this course segment or continue in it as long as he desires. It is possible for someone to sign on, complete six successive clinical problems, and be finished in as little as two minutes. (The value that we require in this scoring counter to demonstrate mastery is dependent on the length, complexity, and nature of the material discussed.)

However, if he misses the one key question in any clinical discussion, his scoring counter is set to zero and he is branched to the corresponding anatomical discussion.

### Anatomical discussions

The anatomical discussions differ from the clinical problems in several important respects:

(1). They are highly branched. For some questions there are as many as 35 anticipated answers with up to ten different branches, depending on which anwser is given. It does not require many such nodal points to produce a highly complex network. It is possible for a student to stay in a single anatomical discussion for 30–40 minutes without retracing his steps. However, it is unlikely for him to have to do so, since hopefully he is learning at every decision point.

(2). For each anatomical discussion there usually is only one starting point, and one logical exit point. Despite the complexity implied above, the entrance and exit points may be adjacent to each other, i.e., it is possible to come in, answer two questions, and be out. In practice this seldom happens, since some subset of the question that permits him to get out is included in the clinical problem which sent him into the anatomical discussion. We simply are following the well known pedagogical axiom that one can only hope to get across one or two major points in a discussion. Some individuals can appreciate these general principles in their barest form, while others need elaboration.

Let us illustrate this with one example. The student misses a clinical problem and enters an anatomical discussion. The first question he is asked may be:

> WHAT IS THE ACTION OF THE RIGHT SUPERIOR RECTUS MUSCLE?

The correct answer, assuming the patient is looking straight ahead to start with is:

> IT MOVES THE EYE SUPERIORLY, MEDIALLY, AND ROTATES IT IN A CLOCKWISE DIRECTION AS YOU FACE THE PATIENT.

This may sound like a fairly difficult question and certainly we obtain a variety of answers. However, the student is shown how to reason out the answer by a series of leading questions and explanatory pictures.

The question that follows the correct answer to the first question, and the one he has to answer to get out of this part of the discussion in essence is: "What would you do with this knowledge?" More specifically he is asked:

> WHAT WOULD YOU ASK A PATIENT TO DO THAT WOULD TEST THE ACTION OF THE RIGHT SUPERIOR RECTUS AND ONLY THIS MUSCLE?

Here is where our challenge lies—to teach the student to question the validity and significance of facts—to train him to reason. What good is it that a physician know the action of a muscle if he cannot utilize this knowledge by testing the muscle in his patient?

(3). If the student entered the anatomical discussion via a clinical problem and reaches this normal exit point, i.e., has answered the above question correctly, he will branch to the next clinical problem and once again try to answer six in a row correctly.

(4). At any point in an anatomical discussion, but at no point in a clinical problem, the student may ask any question he wants. He is then branched to a subroutine which analyzes student questions.

### Student questions

At any point in an anatomical discussion when he is asked a question he may choose not to answer it, but rather to ask a question of his own. His motivation may be that he thinks his own question will lead him to the answer he is lacking, or he may in effect be saying: "Okay, I've had enough of this particular line of conversation, let's proceed to something I don't already understand." Whenever he asks a question three things are permanently recorded: his name, his question, and where in the program he asked the question. The address of the question he avoided answering is stored so he may be returned to this point.

His question is first prescanned (key letter analysis) in order to determine whether it is germane. If not, he is told so and returned immediately to the question he avoided answering. However, if his question is germane, it is further analyzed and he is branched to some other point in that anatomical discussion or into another module, where he is shown how to reason out the answer to his question. We prefer this to giving direct answers to his questions. If he is branched to a place where his question is answered immediately, the reasoning behind this answer and a probing analysis follow.

Once the student is in the question asking routine, and after his question has been answered, he has several options open to him. (1) He may continue to ask as many questions as he likes, thus branching from point to point within a given anatomical module or, more commonly, branching from one anatomical discussion to another. (2) He may signal the computer at any time that he is ready to return to the point where he asked his *initial* question. (3) He may, without knowing it, reach the normal exit point of an anatomical discussion. However, since he is in the question asking mode, he is treated differently than if he had entered via a clinical problem. Instead of being branched to another clinical problem, he is returned to the point where he asked his *initial* question. Thus there is no way he can avoid the question he originally chose not to answer.

Remember that when he first signed on the course segment he was given an option as to whether he wanted to ask a question or whether he wanted the computer to initiate the discussion. If he chose to ask a question at that time, he would have entered exactly the same subroutine. He would have been handled in the same manner with the following minor exceptions. If he reached the normal exit point of an anatomical discussion, he would be branched back to his starting point and given the option again. If he signalled the computer that he had tired of asking questions, he would in essence be saying that he wanted the computer to take the initiative and would then be branched to a randomly selected clinical problem.

There are several distinct advantages to the experimental format currently being used by our students.

(1). Authoring is greatly facilitated by the use of modular units for course construction. It is one thing to sit down and write a lecture or linear presentation, but quite another to outline a highly branched, open-ended discussion. The smaller the modules, the easier this is to perform.

(2). Relevance and interest are maintained through the "clinical problem" approach to human anatomy. The clinical problems, however, are just one type of application question which is common to many disciplines. They provide a certain amount of interest or spice to the learning of what otherwise might appear to be a series of facts or skills which often seem irrelevant. The question of "relevance" is even more important than providing interest. There is more to learn than we have time to teach and sometimes we, as teachers, tend to get carried away by details that happen to have special interest for us. Thus the application questions help to keep *us* "honest" and relevant. *If* a piece of anatomical knowledge cannot be accessed via a clinical discussion, perhaps we should question its significance to the student.

(3). The ability of the student to ask free format questions and be shown how to reason out the answers, gives him the feeling of being treated as an individual. He can literally chart his own path through a discussion until he is ready to be evaluated, i.e., to enter the clinical problems. Teaching the ability to ask questions and to reason out the answers is one of the most difficult tasks we face as teachers.

(4). The high ratio of clinical problems to anatomical discussions, the redundancy and highly branched nature of the anatomical discussions themselves, and the ability of the student to ask free format questions, all contribute in permitting students taking the same course segment to have relatively unique experiences. Not only do they get different clinical problems, but they may not even be taken to the same anatomical discussions. We find that this variety of experience *inside* the classroom stimulates discussion *outside* the classroom.

(5). It is the combination of the features discussed above that permits one student to be told he has mastered the material in one course segment in as little as two minutes, while another student may spend several hours to attain the same degree of mastery. This raises the interesting implication of informing a student when he has attained sufficient mastery of the entire subject matter, rather than giving him a course grade. Some students, either because of ability or previous experience, might achieve this level of mastery in a month, while another student might require the present six months. The faster students would have a lot of free time which could be spent on other courses, independent study, electives, or research. Thus it is conceivable that once a curriculum were implemented on the computer, the student's medical college transcript might more meaningfully consist of a record of how many things he accomplished during his training, rather than a series of numerical grades.

### *Artificial intelligence*

This may be a rather grandiose term for the rather primitive examples we have, but we want to discuss two general topics, i.e., improving the methods by which we handle the student questions, and developing self-adaptive programs.

## Student questions

As a result of experience with students on the initial course segments, we found that a large percentage of the questions they asked either were not answered or were not handled appropriately. This does not have to happen very often to discourage a student from asking any further questions. However, we recorded every question a student asked, so we were able to review them. We found there were three main reasons for mismatches on the questions: (1) the question was not related to the subject matter being discussed, (2) the student did not provide enough information, and (3) he provided too much information.

(1). If the question is not pertinent to the subject being discussed, we have no need to answer it. This was determined by prescanning for keywords. We found, however, that we could not always tell whether the question was not pertinent, or whether it was just not specific enough. Basically we solved this by equating certain synonymous terms and by adding to the number of keywords in the prescan. We also added two other levels of scanning. The first is for such things as leg, arm, thorax, etc., which are parts of the body far removed from the eye. If these are detected, the student is told, for example:

> WE ARE NOT DISCUSSING THE LEG AT THIS TIME. PLEASE LIMIT YOUR QUESTIONS TO THE SUBJECT UNDER DISCUSSION.

We have a second level which includes keywords related to the region, but not to the subject. Thus if his question referred to the maxillary nerve, part of which does run through the orbit, he would be told:

> THE MAXILLARY NERVE IS RELATED TO THE ORBIT, BUT DOES NOT INNERVATE ANY OCULAR MUSCLES.

If he did not match on any of these three levels of prescanning, he would simply be told:

> YOUR QUESTION DOES NOT APPEAR TO BE WITHIN THE SCOPE OF OUR DISCUSSION. DO YOU WANT TO REPHRASE IT?

If he does not choose to rephrase it, he is branched back to where he asked the question. Differentiating whether his question is not germane or whether it is not specific enough is almost essential. Trying to determine in what way it is not related simply makes the dialogue a little more personal and gives the student the feeling he is being treated as an individual.

(2). The most common difficulty was that the student did not supply us with enough keywords, i.e., his question was not specific enough. Thus we have developed a little subroutine which helps him make his question more specific. For example, if the only keyword we detect is MUSCLE, we ask him:

> WHICH MUSCLE OF THE EYE AND WHAT DO YOU WANT TO KNOW ABOUT IT?

He then is given the chance to rephrase his question. Thus with relatively little programming we can interact with the student in a conversational manner until his question is understood. On the basis of previous experience we feel we will be able to handle most of the questions asked.

(3). The third area where we sometimes had difficulty was when the student provided us with too many keywords. It is a surprising fact that the number of keywords required in a given course segment to provide us with enough information to answer a question is remarkably constant. In the program on the muscles of the eye it was three. When there were too many keywords, analysis showed he was usually asking more than one question, or at least what he thought was a single question could be broken down into two smaller ones. Less frequently he was simply being too verbose. Formerly he would branch on the basis of the first three words that matched, but this was not always appropriate. Now we count and store the number of keywords in his question. If this exceeds our magic number, in this case three, the words we have detected can be displayed for him on the screen. He then is asked to rephrase his question using no more than three of these words, or to ask only one question at a time.

## Self-adaptive programming

We would like the program to modify itself on the basis of experience, much as a teacher learns from his experience with students. As a result of our own research in neurophysiology, we feel that two basic aspects of learning are: (A) an increase in seeking or exploratory behavior following cessation of a rewarding stimulus,[1,2] and (B) habituation or the dropping out of unrewarded components of a response.[3] A teacher, at least a good teacher, when challenged is ready to increase the variety of his response. This is an example of exploratory behavior. He may do this by retrying responses that were previously part of his repertoire, but had been temporarily discarded, i.e., had undergone habituation. He may also increase his repertoire of response by incorporating responses acquired from experience with students. At the present time we have only begun to incorporate these learning concepts into instructional programs.

The following are examples of capabilities we consider necessary for the computer if it is to approach the versatility required in tutorial discussions. The first two exist only as isolated demonstrations at selected points, because COURSEWRITER II does not permit us the computational ability to do this on a large scale. The third example, which we consider of utmost importance, has not actually been implemented as yet, but we foresee no major obstacles, except for the limited computational capacity of the system.

(1). If a certain percentage of the students (currently 20 percent) all ask the same question at the same point in the program, subsequent students are branched as though they had asked the same question. They are treated as though they were in the question asking mode, e.g., when they reach the normal exit point of an anatomical discussion, they are returned to the point where they came from. This branching is dynamic and reversible in the sense that the need for asking the question is constantly evaluated. Thus if two students in any series of ten ask the same question at the same point, every odd-numbered student that follows is branched as though he had asked the question. Even-numbered students are not branched. If nine of the next ten even-numbered students fail to ask the question, the branch is deleted. However, if two or more of them do, then the branch is reinforced, i.e., three out of every four successive students will be branched. Certainly if a significant number of students did ask the same question at the same point in a discussion, we as teachers would probably modify our approach. How often this will occur and whether the percentage should be greater or less than 20 percent are questions we cannot answer until we can test it on a larger scale.

(2). There often are several places in a program to which we could branch a student in response to his question. At present we make this choice for the students. We plan to give them some degree of control by forming a hierarchy of possible branch points. Originally these will be evaluated by us as first, second, third, or fourth choices. However, each time a student is branched and reaches the point where we think his question should have been answered, he will in effect be asked: "Okay?" or "Does that answer your question?" If he says yes, the likelihood of that branch will be augmented. If he says no, he will be branched to another point and the likelihood of the original branch will be decremented. Thus what we thought was the least plausible response to a given question may be shown to be the most desirable on the basis of experience with students, and it will achieve the status of the initial branch without any manual interference by the author.

(3). One of the most significant ways a teacher learns from experience with his students concerns the unanticipated but appropriate answer. Right now we record all unanticipated answers and review them periodically. Occasionally an unanticipated answer proves to be more perceptive than the anticipated answers the author programmed. At present such a student is treated as though he were wrong.

When a student gives an unanticipated answer and feels he is treated in an inappropriate manner, why not permit him the option of repeating his answer and treating it much as we would a question? Essentially he would be entering a "debate mode". We feel that our question answering routine is sufficiently flexible now that he would eventually be taken to a point where he could decide whether his original answer was valid or not. If it proved invalid, he would be branched back and his pathway erased. However, if he felt he had won his point, then his route could be preserved. This would then become an anticipated answer for subsequent students. In interpersonal discussions our students often challenge us and not infrequently they win their point. However, even if this occurred only once in a thousand times, these are the type of responses we would least like to discourage. How can we profess to encourage our students to question and reason and then give an inflexible response? This is a level where computers are not presently competitive with a human tutor.

Since we have not yet implemented this, and do not want to be considered idle dreamers, we shall elaborate on how we intend to program this type of ability. First it should be made clear that we are not talking about situations where the student's response involves evidence not available in the program. We are talking about situations where he reasons from one logical statement to another. Let us cite a specific example. In our original version of the discussion of the eye, we programmed many anticipated answers to the question:

WHAT IS THE ACTION OF THE SUPERIOR RECTUS?

One answer we did not program was:

THAT DEPENDS ON THE STARTING POSITION OF THE EYE.

We subsequently modified the program to include this as an anticipated answer. However, the inherent logic was already present for the student to have won his point. If he had asked—in debate mode:

WHAT IS ITS ACTION IF THE PATIENT STARTS BY LOOKING MEDIALLY?

he would have been given one answer. If he then asked:

WHAT WOULD ITS ACTION BE IF THE PATIENT STARTS BY LOOKING LATERALLY?

he would have been given a very different answer. Clearly this would prove that the action depends on the starting position of the eye.

The computer has no such ability to reason, but the student does. Thus we are permitting him to make value judgment. He could signal the computer that these two answers made his point and subsequent students would then branch there, rather than along the path previously followed. Since we are permitting the student to make a value judgment that affects the subsequent course of his fellows, the process must be reversible. Thus the next ten students who gave the same answer would be asked by the program whether they understood the line of reasoning that followed. If the consensus were yes, then the branch would remain; if it were no, then the branch would be deleted.

*Interactive graphics*

Gross anatomy is very largely a visual science. Knowing the three dimensional relationship of one structure to another is a fundamental basis for clinical diagnosis. The best way to organize this information is with pictures, so our students are encouraged to spend a lot of their time sketching. In our linear (non-computer) programmed teaching they can actually sit and copy pictures that are projected. The question then arises, does the computer offer interactive graphic capacities that are competitive? In order to explain what we have done and our problems in this area, it will be necessary to go into some of the details of the system with which we work, since it is quite different from what most people think of when they speak of CRT graphics.

The 1510, which is the CRT, light pen and keyboard unit, was designed primarily for the rapid display of text, and its designers assumed that its graphic applications would be limited. The usable area on the face of the CRT is 4¾ × 8 inches. It may be thought of as a grid consisting of 32 rows and 40 columns (Figure 2). A standard alphanumeric character would occupy two of these boxes, i.e., two rows by one column. Each box on this grid, i.e., each one row by one column unit, may be thought of as a matrix of 48 potential dots of light, six dots high and eight dots wide (Figure 3). Thus the entire screen consists of a maximum of 61,440 dots (192 vertical × 320 horizontal). Actually these are more accurately described as horizontal slashes; the dots are wider than they are high as may be seen in Figure 4. This is a significant factor which must be considered in preparing the drawings, to prevent distortion.

The system provides a standard character dictionary and the user may define additional graphic sets. These graphic characters, as defined by the system, occupy such a large part of the screen that the likelihood of being able to use the same graphic character to con-

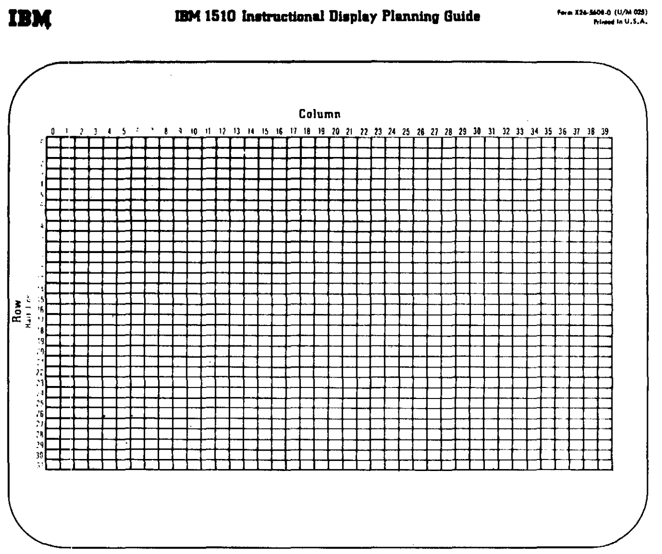


Figure 2—This shows the organization of the screen into 32 rows and 40 columns forming 1280 addressable units. The standard alphameric characters occupy two of these boxes, i.e., two rows by one column

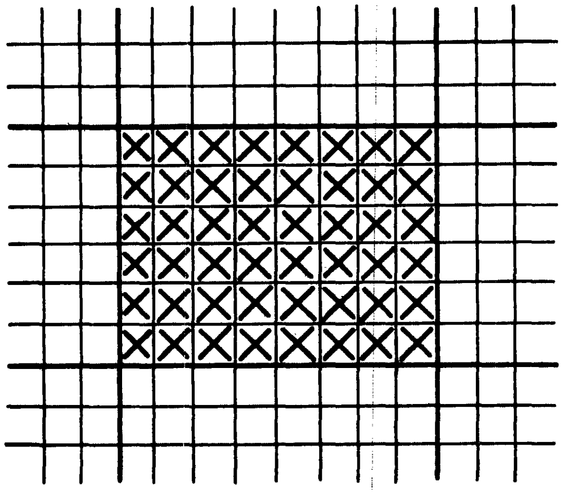


Figure 3—The 48 dot matrix defined by the intersection of one row and one column

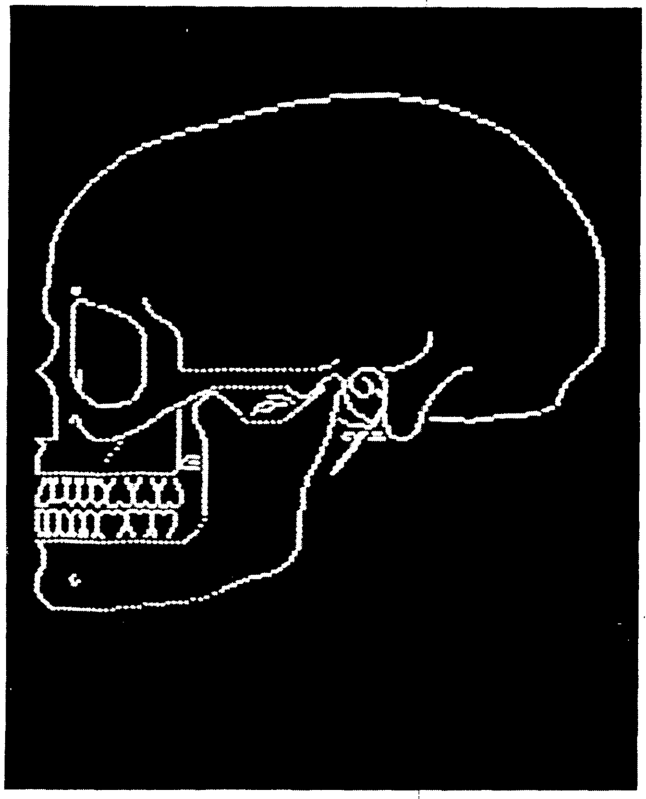
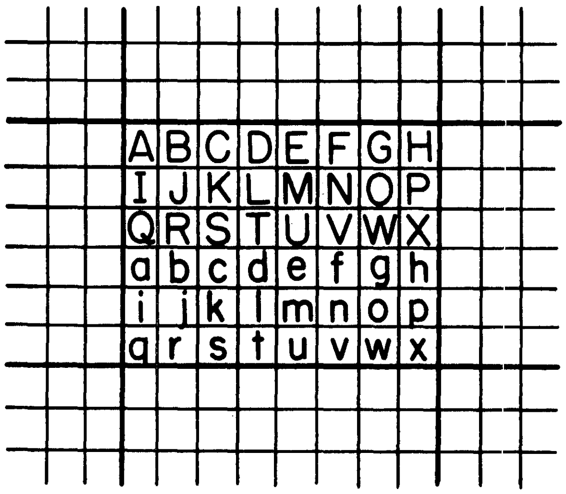


Figure 4—A CRT display of the skull and mandible from the side

struct more than one picture is almost nil. It is analogous to taking a printed page, dividing it into four quadrants and saying you can use these quadrant units to write anything else you desire, as long as you don't change any of the letters or words. It quickly became obvious that we did not want graphic characters of the type just defined, but rather we needed a graphic *alphabet*. Just as in the case of the English language, given the 26 letters of the alphabet, one can write anything he likes, so given the means to directly access each of the 48 dots in each box, we could draw any pictures we desired.

That is basically what we did; we defined a character dictionary with each character being a single dot (Figure 5). Thus with what amounts to little more than 1/3 of one character dictionary area we can draw as many pictures as we desire. The backspace function permits superimposition of characters. Thus if our display instruction were to contain the following characters, as defined in our graphic alphabet, i.e., BCDEFJNRVbfjklmn, and there were a backspace command between each of them except the "m" and the "n," we would get the dot pattern shown in Figure 6. Notice that omission of the backspace instruction caused the "n" dot to appear in the six by eight box one column to the right.

We always limit our display instructions to one row at a time and we put as many instructions on each row as possible, i.e., we try to break our pictures up into the smallest units we can. This permits us greater freedom with the input buffer (250 character limit), facilitates debugging, allows us to modify pictures with a maximum of ease, provides animation capacity, and is especially useful when we give the student the capacity to draw his own pictures. However, there is at present one very serious limitation to putting multiple



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H |
| I | J | K | L | M | N | O | P |
| Q | R | S | T | U | V | W | X |
| a | b | c | d | e | f | g | h |
| i | j | k | l | m | n | o | p |
| q | r | s | t | u | v | w | x |

Figure 5—The characters used to define our graphic alphabet. The character plus its case determines the position of the dot within the matrix

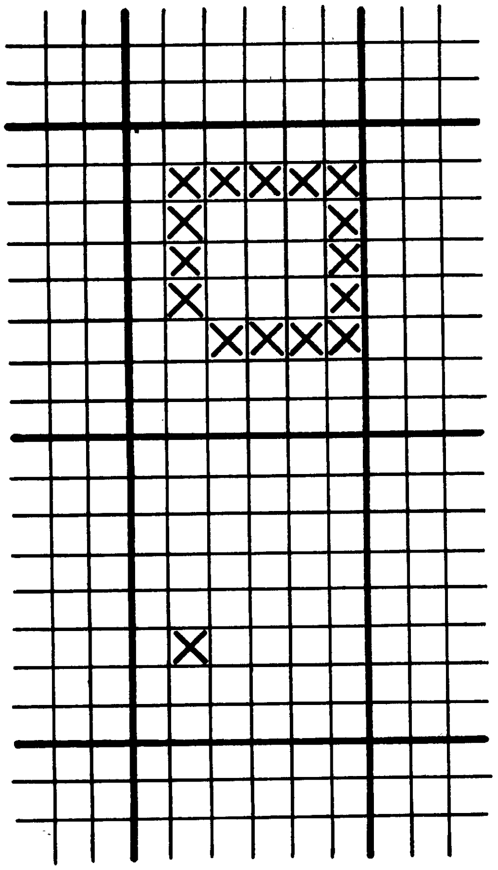


Figure 6—The pattern produced by the following coding <B%C%D%E%F%J%N%R%V>%B%F%J%K%L%MN The "<" defines subseqnent characters as upper case. The ">" defines subsequent characters as lower case. The "%" is the code for the backspace function

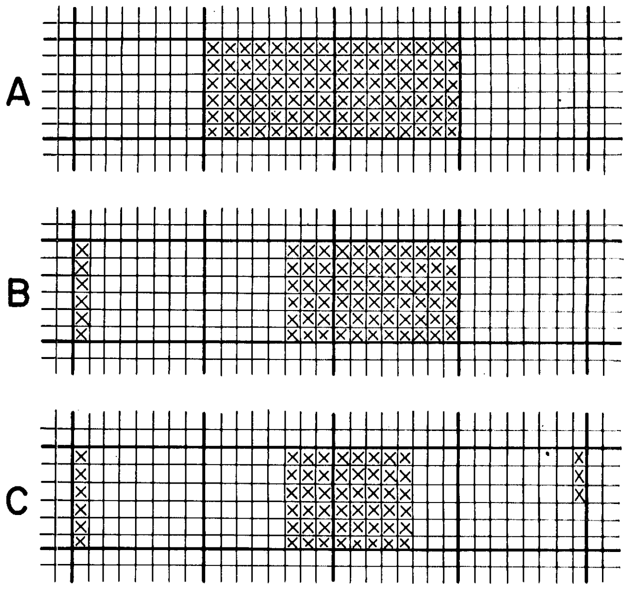


Figure 7—This shows the problem of erasure with multiple display inserts on the same row. The insertion of the single dot column in (B) causes erasure of 5 dot columns from the original pattern in (A). Insertion of the 1/2 dot column in (C) causes erasure of 3 dot column from the original pattern

display instructions on a single row. No erasure occurs between adjacent rows, i.e., the upper and lower limits of each box are inviolate. However, erasure does occur between adjacent columns. Let us assume that we had two adjacent boxes on the same row filled completely with dots [Figure 7 (A)]. If subsequently any pattern were displayed on the same row in the column just to the left of this, e.g., a vertical line in the extreme left of the box, we would get the pattern shown in Figure 7 (B). A subsequent display instruction on the same row but in the column just to the right of our original display, e.g., three dots vertically arranged in the extreme right of the box, we would get the pattern shown in Figure 7 (C). Thus a display insert command erases five dot columns to the right of the insert and three dot columns to the left on the same row. We have been told that this can be improved on a hardware level so no erasure will occur. This would be of utmost importance to anyone who wants to exploit the graphic capacities, especially in having the student draw on the CRT.

The resolution of the light pen is limited to one box as defined by one row, one column. Light detected from one box can be differentiated from light in any other such box. Two lighted dots are required to produce a detect and these two dots must be separated by one dot row. Thus the pattern's shown in Figure 8 (A) would all permit detection; those in Figure 8 (B) would not.

We use the light pen as a pointer. We have not been able to devise any means of using it as a stylus, although we do have various ways in which we can have the student draw on the CRT. Some of the ways we use CRT graphics are enumerated and briefly described.

## Identification

In the CRT display shown in Figure 9, we have the student use the light pen to identify the structures labeled in Figure 10. We feel that since we are dealing with a picture approximately four inches in height, this is pretty good resolution. As with verbal questions, we branch selectively not only according to whether he is right or wrong, but also on the basis of what the nature of his error is. Thus his thinking is analyzed

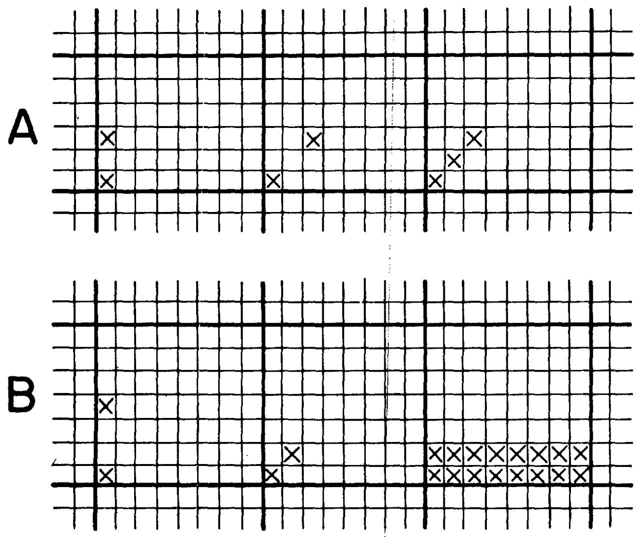


Figure 8—(A) shows three dot patterns which permit light pen detection; (B) shows three dot patterns which would not be detected by the light pen

and he is led by discussion or demonstration to the correct answer. Since the face of the CRT is behind a glass cover, we have a parallax problem. The boxes that he is trying to define measure only 1/6 × 1/5 inches. We cannot vary the intensity of the beam by tracking the pen. We can require a double detect, i.e.,

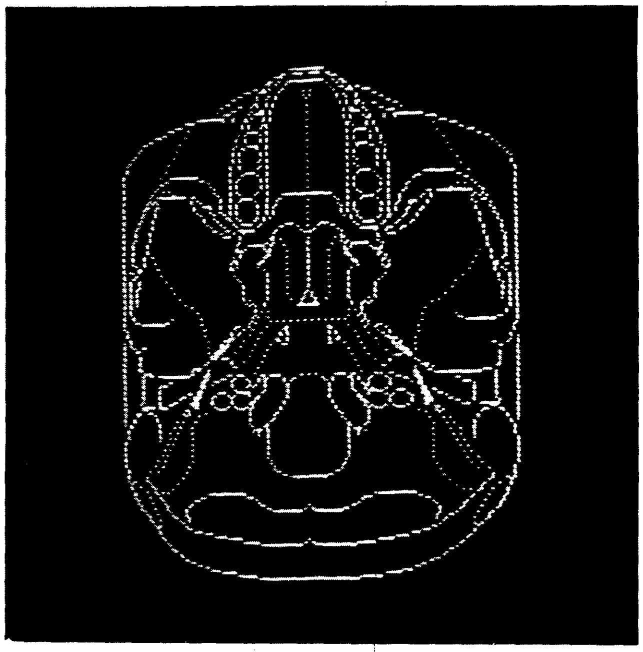


Figure 9—A CRT display of the base of the skull

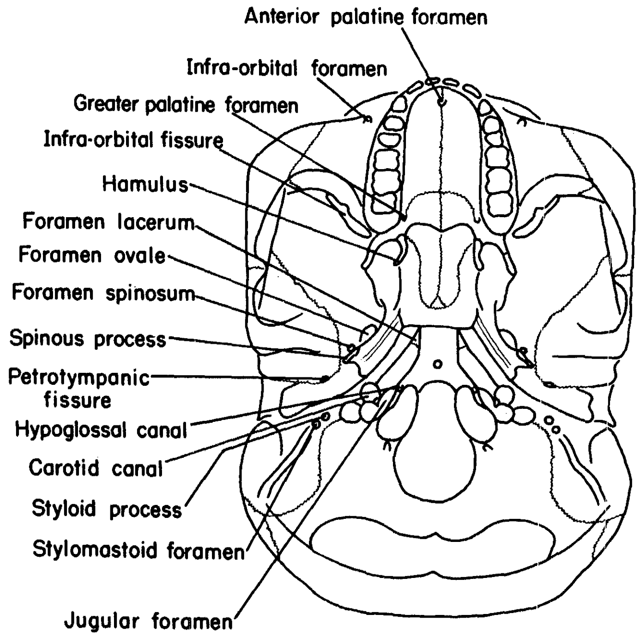


Figure 10—This shows some of the structures that we require the student to identify using the light pen on the graphic display shown in Figure 9

on the first detect temporarily erase the adjacent boxes and ask in effect: "Is this what you want to point to?" However, in practice we do not find this necessary. After a little experience the students make very few parallax errors.

## Animation

We use a few examples of animation in the usual sense such as moving the eyes, swallowing, etc., which can be done in the insert mode. This is quite effective as long as only part of the picture has to be regenerated. More commonly we employ animation in the sense of drawing something slowly for purposes of emphasis. For example, when we ask a student to point to where a nerve originates, after he does so correctly, we may respond by having the nerve "grow" out along its course.

## Enlargement

The 1510 has no vector or scaling capacities. However, we do present a small scale view of a structure such as the skull and then enlarge certain parts of it in 2X steps until we get the desired resolution for light pen interaction or to show greater detail.

## "Drawing" on the CRT

The quotation marks are to emphasize that the light pen cannot be used as a stylus. This would be desirable, of course. However, this is not as great a limitation as it might seem, since we are trying to get the students to appreciate spatial relationships and proportions, rather than training them as artists. There are several means by which we permit students to generate their own pictures and have them evaluated. In each of these instances, the erase feature is a distinct limitation, and we are actually delaying much of our development in graphic until a hardware modification comes through.

(1). We present the student with our dot matrix and have him input from the keyboard, evaluating his picture segment by segment. This may sound artificial but it works quite well. However, from the keyboard there is a 100 character input buffer, so here, more than anywhere else, we feel the limitation of the erase feature.

(2). We put a lighted square in each box. The student has three modes of operation from which to choose. If he is in the insert mode, touching a lighted box causes the square to be replaced by an asterisk like symbol. The replace mode causes the square to replace the asterisk, e.g., if he changes his mind. When he is finished he enters the erase mode in which every square he touches disappears and he is left to view his finished drawing [Figure 11 (A)]. The drawing is then evaluated by the computer, and those parts of his drawing that are judged to be accurate are regenerated using our graphic alphabet. Thus his drawing, represented in Figure 11 (A), would be presented back to him as in Figure 11 (B). However, any parts of his drawing not judged accurate would be left alone and he would have to try again. A photograph of this view of the skull is shown in Figure 12.

(3). We have every bone in the body drawn on coordinate paper. On the CRT a graph paper grid provides the lighted matrix for the light pen detect. In essence we have him point to a series of points and if he is correct, we generate the line of appropriate contour between successive points. With soft tissues, e.g., organs, muscles, etc., we are concerned with their relation to bones. The bony skeleton then becomes the lighted matrix upon which he draws. For example, it is of vital importance that the student know the normal projections of the heart and its various subdivisions onto the thoracic cage from every angle. Thus we present him with a graphic of the bony rib cage and ask him to point to where each chamber or structure crosses the bones, and generate the pictures as he progresses.

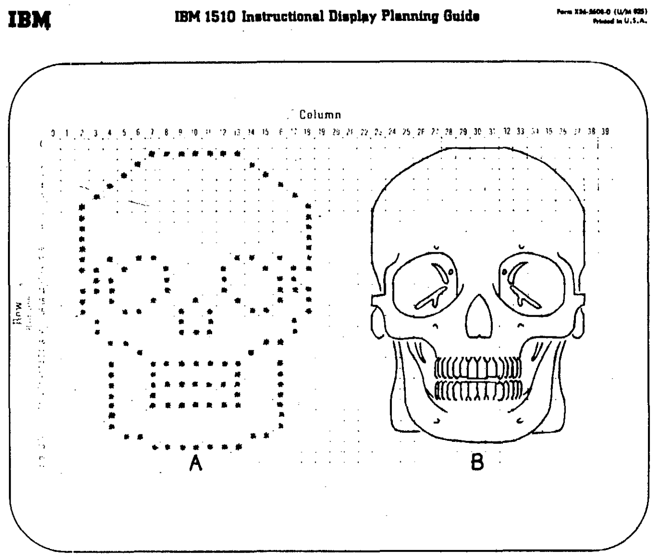


Figure 11—(A) shows a crude form of light pen drawing by the student; (B) represents the computer evaluation of the drawing using our graphic alphabet

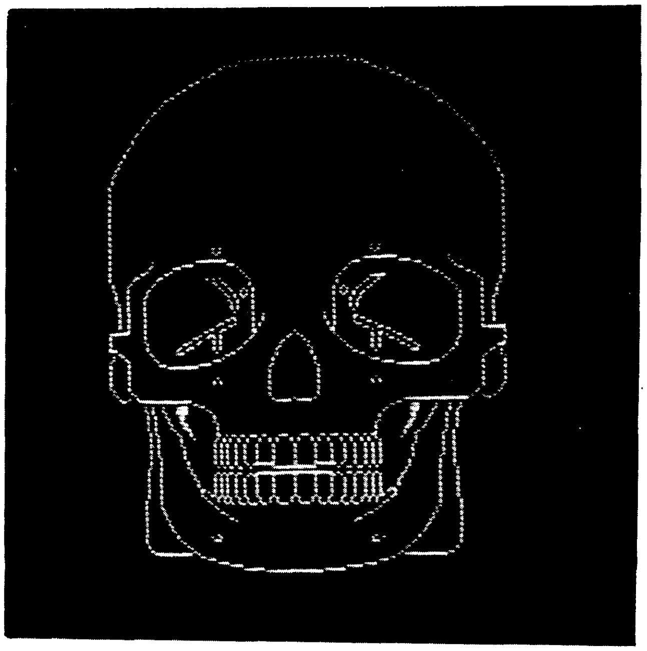


Figure 12—A CRT display of the skull from the front

## SUMMARY

We have tried to describe some of the natural language analysis, self-adaptive programming, and interactive graphic capabilities we feel are required for medical education. Although the system and language we have been using were designed for CAI, they were not designed for the further capacities toward which we have tried to force them. We would like to have a system and a language that were tailored to meet the needs of higher education.

CAI is expensive, but so is medical education in its present form. Any tool that would significantly improve the quality of medical education can hardly be denied on the basis of cost. The real question is whether CAI can justify itself on a *performance* basis. Perhaps in two, five, or ten years the computer industry will feel the state of the art justifies a real commitment to this field. However, will what they produce truly meet the needs of the medical educator unless a really interdisciplinary phase of research and development is undertaken now?

## ACKNOWLEDGMENTS

The authors gratefully acknowledge the help and encouragement of the many people at The IBM Systems Research Institute.

This investigation was supported by General Research Support Grant FR-05396 from the General Research Support Branch, Bureau of Health Professions and Manpower Training, National Institutes of Health and by Grant #50/68 from the National Fund for Medical Education.

## REFERENCES

1 W D HAGAMEN
*Responses of cats to tactile and noxious stimuli: Temporal summation, facilitation, internal inhibition, and external inhibition as examples of interactions between stimuli on a behavioral level*
Arch Neurol Vol 1 1959 203-215

2 N F O'DONOHUE W D HAGAMEN
*A map of the cat brain for regions producing self-stimulation and unilateral inattention*
Brain Research Vol 5 1967 289-305

3 S L JAFFE P F BOURLIER W D HAGAMEN
*Adaptation of evoked auditory potentials: A midbrain through frontal lobe map in the unanesthetized cat*
Brain Research Vol 14 1969 111-127

# Design principles for processor maintainability in real-time systems

*by* H. Y. CHANG and J. M. SCANLON

*Bell Telephone Laboratories*
Naperville, Illinois

## INTRODUCTION

With the arrival of large real-time, time-shared systems, the requirement of system reliability has become even more demanding. The result of even a momentary system misbehavior could be catastrophic, since any disruption of service is experienced by all the users on-line at that time. Thus for real-time systems such as telephone switching systems, airline reservation systems, on-line teaching machine, etc., where numerous users are served, and critical real-time systems such as command and control, a high degree of system dependability and maintainability must be realized.

Since many of the real-time systems employ the concept of centralization of logic, the overall system reliability objective in large part depends on how well the central processor itself meets the dependability and maintainability objectives. For a processor, the dependability objective often calls for the use of reliable components, conservative circuit design techniques and various redundancy methods. The maintainability objective, on the other hand, demands a processor architecture that is best suited for automatic trouble detection, recovery from faults and fault isolation, so as to insure operational survivability in an environment which is not free of faults.

The purpose of this paper is to describe several design principles which may be used in planning processor organization, designing logic circuits, and fault detection and diagnostic tests in order to facilitate the design of a highly maintainable processor for real-time systems. Our scope will be limited to presenting a unified account of some design guidelines, most of which reflect material assembled from a combination of analytical study and practical experience on a real-time time-shared system.[1] The problem of achieving high dependability by the use of various error detection and correction codes or redundancy techniques has received adequate treatment in the literature,[2–4] and will not be included here. In the second section we describe the various observed trouble symptoms and their manifestation in the system. A maintenance sequence for preserving the system's integrity upon occurrence of faults is then suggested. Guidelines for planning a processor organization to achieve high maintainability are discussed in the third section. Several principles for designing logic circuits and fault detection and diagnostic tests are described in the fourth section.

### *System malfunctions and recovery procedures*

An important first step in establishing a fault recovery and detection philosophy for a particular system is to establish the possible failure modes of both system and device components. On a system level, trouble symptoms usually manifest themselves in some form of mutilated data. They can be caused by errors in transmission or reception of data among the various units; e.g., a bit erroneously set on a memory access. Or, they can result from errors in internal data manipulation, e.g., attempting to reach an address which has been incorrectly computed.

On a device level, the trouble symptoms with discrete logic implementation usually correspond to single, hard faults (by common assumption). A permanently open diode and a transistor output stuck-at-1 (s-a-1), are some examples of this class of faults. However, these troubles usually manifest themselves in some observable system malfunction. With the advent of integrated circuit technology, more complex and varied device failure modes may be expected.

As one of the principal requirements in a real-time facility is to provide continuous service, the system must remain operational even in a fault environment. This dictates that trouble symptoms be recognized and the associated fault be isolated and repaired, with little or no interference from the user's standpoint. This objective can be implemented by devising a fault recovery procedure. A fault recovery procedure usually consists of the following steps: fault detection, fault recognition, system recovery, and fault diagnosis.

*Fault detection* is usually a function entirely performed by a variety of hardware implementing error detection codes such as parity checks, one-out-of-N codes, etc., and analogue signal margin checkers. Systems incorporating some level of redundancy may also use matching between duplicated modules as a means of fault detection. In all cases the checker itself should routinely be examined by programs to insure its validity.

The objective of *fault recognition* is to resolve a failure to a particular subsystem (e.g., a memory module, an input/output channel controller etc.). This is done by first establishing the type of error which has occurred such as a parity failure on a memory read, and determining from that information, through some analysis procedure, what subsystem contains the fault. The analysis procedure may include a sequence of instruction retrys in order to distinguish the hard faults from the transients, and then to resolve the failure to the subsystem level by alternately exercising various suspected candidates. It may also examine subsystem error indicators, over some period of time, to accumulate clues pointing to the source of malfunction.

Once the failure is resolved to a subsystem, choosing the next step in the fault recovery procedure depends upon whether or not a spare subsystem is available.

If a spare is not available, *diagnostic* action must be initiated to determine the identity and location of the fault. The normal system operation, which had momentarily been interrupted at the time of fault detection, must now be suspended through diagnosis and repair. The system must then be *recovered* to a hardware state and program point where normal processing can be resumed. This sequence of events is depicted by Figure 1.

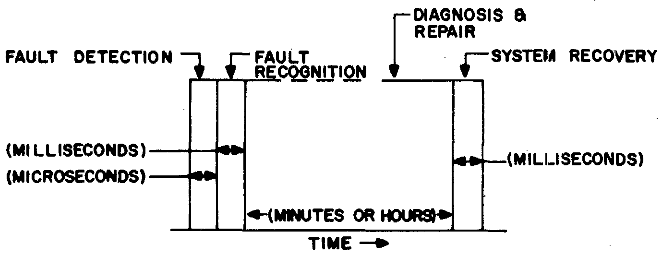


Figure 1—Fault recovery sequence (without spare)

However, if a spare is available, a different strategy could be taken. The system is first reconfigured by interchanging the faulty subsystem with its corresponding spare, using some method of program controlled switching.[1] The *recovery* procedure is then initiated to restore the system to a normal processing state, in order to reduce the period of interrupted service. The task of diagnosis and repair can be postponed and offered to the system as a relatively low priority job since it is the most time consuming step of the recovery procedure. This sequence of events is depicted in Figure 2.

A comparison of Figures 1 and 2 illustrates some of the maintenance advantages of hardware redundancy. First the diagnostic task, which generally consumes more time than all the other recovery steps combined, can be deferred and interleaved with normal system processing on a time-shared basis after the system is restored to sanity. Secondly, the availability of a spare permits a "good" vs. "bad" comparison type of diagnostic testing where the "good" machine inter-

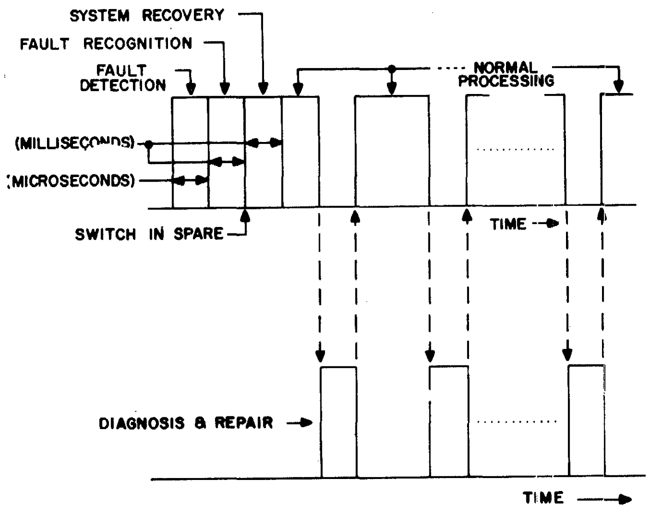


Figure 2—Fault recovery sequence (with spare)

rogates the faulty machine. This type of testing is readily programmed because of the availability of a spare and hence can be automatic. Without some level of redundancy, an approach must be used whereby the operator acts as the interrogator. This implies manually forcing the machine through recovery steps as illustrated in Figure 1. However, in practice it is often advisable to provide some subsystems with spares, and some without, to arrive at a balance of cost versus reliability.

In most applications, the central processor, whether under program control or some combination of manual and program control, acts as the executor of any system recovery scheme. Thus it is of paramount importance that the central processor itself be highly maintainable. With this in mind then, the remainder of this paper will concentrate on outlining maintenance design principles for the central processor, regardless of the system environment in which it must perform.

*Structural considerations for processor maintainability*

Past experience has indicated that the effectiveness of programmed testing depends not merely on the techniques used in deriving tests and test results, but also on the inherent structural maintainability of the central processor unit. The central processor maintainability is generally constrained by such factors as the modularity of the logic organization, the availability of accessible tests points, etc. It is, therefore, appropriate to list some of the desirable guidelines to be included for consideration in order to achieve overall processor maintainability.

## Modularization

In planning a processor organization for maintainability, modularization is of utmost importance. The processor should be composed of well defined functional modules, with a minimum number of intermodular feedbacks.* This is desirable to confine the effects of malfunctions as well as to facilitate programmed testing. Specifically: (a) the function of each module should be definable as a register, decoder, sequencer, etc. Irregularities such as scattered special flip-flops imbedded into a well-defined decoder or sequencer, or circuits with a mixed mode of synchronous and asynchronous operations should be avoided. The symmetry and the regularity exhibited by the structure of these modules often imply uniformity in the trouble symptoms caused by faults in these modules. As a result, a considerable amount of effort in designing tests and deriving test results can be saved. For example, an attempt should be made to keep general purpose registers logically equivalent so that a single set of diagnostic tests will be applicable to all registers; (b) the interface between modules should be "controllable" and be as simple as possible. This implies that the number of intermodular feedbacks be minimized and that a uniform and consistent method of controlling information flow between modules be established. A common practice in designing tests for a large processor is to treat each functional module individually. As a result it is usually difficult to foresee global problems created by interaction among modules. Many of these interactions can lead to inconsistent test results, i.e., test results that may change from diagnosis to diagnosis.[5] For example, a fault in module A may prevent the initialization of some circuits in module B. If the test(s) for detecting this fault, due to the presence of global feedbacks, also depend on the proper initialization of these circuits in module B, the test results become inconsistent. In a large processor with many functional modules, the testing problems created by these interactions can be extremely complicated. Thus, a "clean" interface between modules is very desirable. This means that in the test mode, every module should be, either directly or indirectly, controllable and monitorable.

## Accessibility and observability

The result of segmentation of a processor into functional modules permits the strategic placement of test points for purposes of controlling and/or monitoring the state of the machine during programmed testing. A method for test point placement has been considered by Ramamoorthy, with the use of graph theory.[6] The functional modules of a processor can be considered to correspond to nodes of a directed graph, and signal paths to edges. The nodes of a graph are partially ordered, from primary inputs to primary outputs. Feedback loops between nodes can be "broken" under the constraint that all nodes remain reachable from primary inputs. Additional control points are then inserted at places where feedbacks have been broken. Test points for monitoring purposes should also be added to modules whose outputs are not observable, either directly or indirectly, at primary outputs. The resultant processor organization is therefore, one in which every module is controllable and monitorable for programmed testing. Consequently,

* An intermodular feedback is a control and/or data path that traverses a ring of functional modules.

the accessibility and observability are greatly improved. Our experience indicates that such a facility can often simplify the design of tests and may well improve the resolvability of faults.

As an example, consider the organization shown in Figure 3(a). Each box represents a functional module. Global feedback loops (BEFB), (CEDC) and (CDC) are broken at edges FB and DC. Every module is still accessible from its primary input (through module A). Control points are added at FB and DC to enable modules B and C. An additional test point is also required at output of module F for monitoring purposes. The resultant organization, with modules partially ordered, is shown in Figure 3(b). Note that every module is accessible from its primary input and/or added control points and the outputs of every module are monitorable at its primary output and/or added test point(s).

As will be seen in a later section, a modular organization with adequate test points will greatly simplify the design of tests. Thus far only the design guidelines for the structure of a processor have been touched upon. Some principles for the behavior aspect are in order.

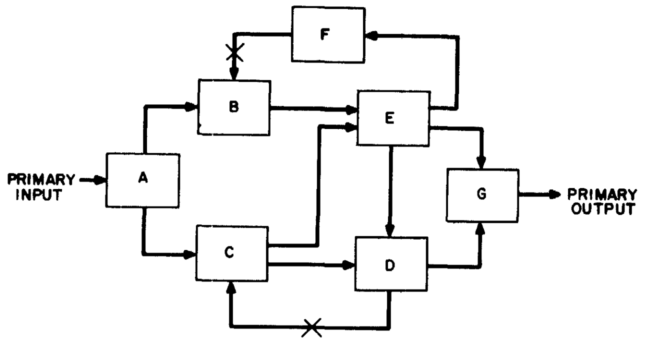


Figure 3a—Functional modules of a processor an example

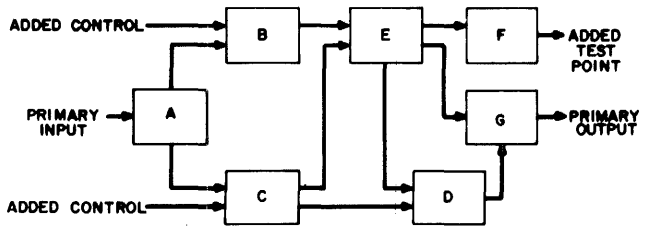


Figure 3b—Partially ordered functional modules of a processor

## Interrupt and rollback mechanisms

As was mentioned earlier, a prime maintainability objective of real-time, time-shared systems is to preserve the system integrity in the presence of faults. The use of error detection and correction circuits may detect and mask out the misbehavior caused by some faults. For example, a system employing a Hamming code can effectively mask out single errors and recognize double errors. However, in real-time operations the tasks of recovery from a fault occurrence usually requires a combination of program and hardware mechanisms. Special interrupt circuitry must be provided which is triggered by fault detection circuits to initiate the recovery process. Protected storage must also be provided to preserve the state of the machine in order to restart the program after the system has been recovered from a hardware failure.

The use of interrupt and rollback mechanisms can be illustrated by the following example (Figure 4). Suppose the normal sequence of operation is $S_1$, $S_2$ . . ., $S_n$ where $S_1$ denotes a steady-state point, or a point to which the program can be rolled back. A fault is detected while the transition from $S_2$ to $S_3$ is being executed. To prevent mutilation of data, this transition should be interrupted and all pertinent information on the state of the machine stored away. The system will then enter a maintenance mode to isolate and repair the fault. Once the trouble is cleared, normal

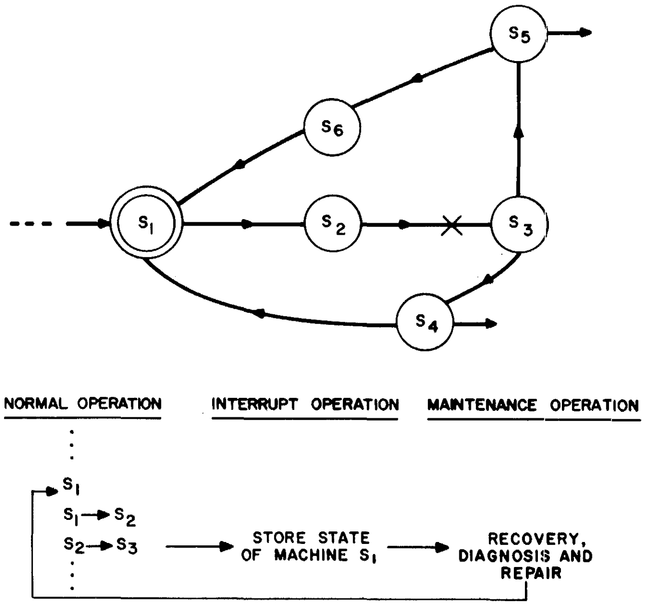


Figure 4—The use of interrupt and rollback mechanisms

operation can then resume by rolling back to steady state $S_1$.

Interrupt and rollback mechanisms have proven to be extremely valuable in real-time operations, especially when there are excessive intermittent troubles in the system.[7] The maintenance of this additional hardware should be made periodically to insure that it is in proper working condition.

### Interface with external devices

In many systems the central processor and its external devices such as memories are interconnected via common buses. To test the circuits of the central processor that are associated with buses, it is often necessary to send data and/or addresses to these external devices. This mode of testing is often inefficient as it requires extensive initialization of devices in the external communities. Furthermore, the test results may be inconsistent since the data is highly dependent on the states of these devices at the time the central processor is to be tested. In a large system a central processor may communicate with numerous devices, and interfacing with these units for testing presents a serious problem. To avoid this situation, a separate return path should be provided (see Figures 5(a) and 5(b)) so that the testing of interface circuits can be simplified. The return path concept establishes a testing environment in which the state of the processor during testing need not be dependent upon the states of other external subsystems or devices. In some cases a saving of twenty to thirty percent of time and program space for testing central processor interface circuits can be achieved.

*Circuit and test design*

Processor maintainability can be greatly facilitated if appropriate design principles are followed in circuit

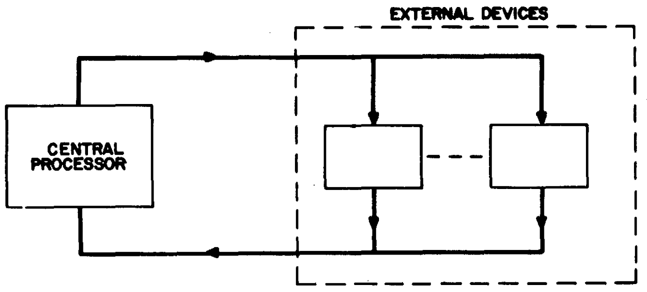


Figure 5a—Processor interface with external system (operational mode)

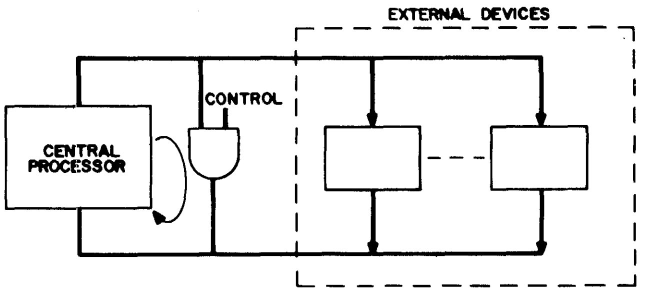


Figure 5b—Processor interface with external system (maintenance mode)

design and in developing diagnostic tests and programs. In this section, we recommend several such techniques, most of which are suggested by our experience and by the results of other workers in the diagnosis field.

### Circuit design

*Circuit Redundancy*—A fundamental assumption shared by all diagnostic methods is the single-fault assumption, i.e., one and only one fault may occur since last diagnosis. The presence of an undetected fault may invalidate this assumption. Consequently, the effectiveness of the diagnostic can be weakened. Thus the requirement of deriving a complete test set, one which is capable of detecting all faults under consideration, is necessary in order to reduce the set of undetected faults that can occur in the field. The presence of redundant circuits greatly complicates the design of detection and diagnostic tests and generally weakens system maintainability. Although faults in redundant circuits may not affect system operation, they could invalidate certain tests designed for other faults under the single-fault assumption.[8] For example, the fault $\alpha$ stuck-at-1 (s-a-1) of a redundant circuit shown in Figure 6 is not detectable. The presence or the absence of fault $\alpha$ s-a-1 has no effect on circuit operations. However, suppose $\alpha$ s-a-1 exists and another fault $\beta$ stuck-at-0 (s-a-0) occurs. The test vector ($x=0$, $y=1$) which was originally designed for detecting $\beta$ s-a-0, is no longer valid, as the path $y \to \beta \to z$ has been "desensitized".[9–11]

As verifying the validity of all tests under all combinations of undetectable or redundant faults is impracticable, circuit redundancy should be eliminated whenever possible.

*Failure Modes*—Many manufacturers have indicated that the use of integrated circuits yields a highly reliable design at low cost. However, to the best of our knowledge the failure modes of integrated circuits

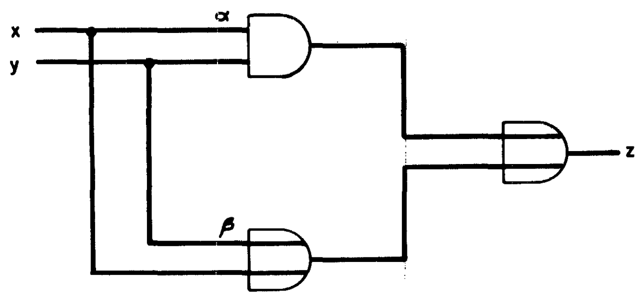


Figure 6—Example of a redundant circuit

and their effect on the test design methods have not been fully explored. From samples that have been studied, the dominant failure modes are still the same as that of discrete components, i.e., stuck-at-1 and stuck-at-0 types. However, there are also many other new modes of failure that may require special attention.[12] Since the integrated circuit configuration can introduce a number of parasitic components (such as diodes and capacitances) between connections, inputs (of NAND gates, for example) can be grounded due to a parasitic diode short. Other modes of failures that are characteristics of physical design include inputs crossing, inputs simultaneously s-a-1 (due to a mechanical bond lifting), collector to emitter short, etc. Until a better understanding of this subject is obtained, one must be cautious in adopting a given integrated circuit for production. A careful study of the feasibility of designing tests for detecting faults exhibiting possible abnormal trouble symptoms should be made.

*Circuit Behavior*—It is generally known that one of the most difficult maintenance tasks is to handle faults, which may be intermittent or marginal, yielding inconsistent failure symptoms. Many of these faults are caused by gradual component deterioration due to aging, manufacturing defects, etc., which are unavoidable. There are others that are caused by overly critical timing, or unrealistically tight tolerances, and can probably be avoided by careful design. Examples of these cases include (a) a hard fault in one circuit which causes marginal operation in another circuit (e.g., hard fault in a voltage regulator), (b) a hard fault in one circuit which prevents the initialization of another circuit (e.g., a fault in a clock gate), (c) faults which cause circuit operation that is dependent upon equipment options employed in the unit being diagnosed, and many others. The test results obtained under these circumstances are usually unpredictable. To avoid diagnostic inconsistencies, the test designers are required to perform the time consuming, arduous task of reviewing the entire unit to uncover these deficiencies, and organizing test sequences by the use of early terminations or selected test skipping techniques. The scope of this task can be minimized if circuit designers are encouraged to design circuits which are well-behaved even under failure. However, since it is unrealistic to assume that diagnostic and circuit designers will be completely successful in preventing marginal or intermittent faults, some tools should be provided to aid maintenance personnel in resolving abnormal fault conditions.

*Connectivity and Packaging*—With the use of large scale integration and the increase in logic density, the relative cost of factory testing and field maintenance is rapidly escalating. Minimizing the number of global feedbacks between modules makes the system less sequential (more combinational); the task of testing, as well as that of generating field maintenance tests and a fault dictionary or catalog, is thus simplified. However, the situation could be further improved if the circuit designer would reduce, wherever possible, the number of fan-ins and fan-outs, and especially, the number of reconvergent fan-outs.* The problem created by reconvergent fan-outs in deriving tests has been noted by many workers.[9,10] It greatly complicates the test generation process and can also affect the fault resolvability, as in many cases faults in fan-out regions are not distinguishable from those in fan-in regions. Thus, reconvergent fan-outs should be avoided, wherever possible.

A common practice in circuit packaging has been to assemble each type of plug-in package to contain several of the same type of logic elements such as flip-flops, p-input NANDS, etc. However, this practice is not necessarily an optimal one from the viewpoint of attaining maximum fault resolvability. As diagnostics are generally associated with "actions" rather than with circuits,[13] serial packaging (i.e., organizing logic elements along paths from inputs to outputs) would yield a far better diagnostic resolvability than parallel packaging. Admittedly, serial packaging will result in more types of plug-in packages. Since in Medium Scale Integration or Large Scale Integration a system may only be composed of several of these packages, the requirement that faults be isolated to one and only one suspected package is quite necessary in order to reduce repair time and/or possible additional unnecessary package replacement. This implies that the

* Suppose gate B is reachable through gate A along some path(s). Reconvergent fan-outs of gate A are those fan-out paths that reconverge at gate B.

use of a serial packaging technique to improve resolution should be carefully considered in the design stage, along with other attributes such as cost of spares, size, quantity, complexity and production yields, etc., to achieve an economic balance.

## Design of maintenance tests

The design objectives of maintenance procedures to enhance processor maintainability are basically twofold: (1) to design a set of tests capable of detecting and isolating all single, solid faults to a replaceable package level, (2) to insure that test results will be consistent for all faults from diagnosis to diagnosis. The aforementioned design principles for processor architecture and circuit design were aimed at facilitating the design and the application of maintenance tests. In this section, we present our recommendations on methods of deriving tests and generating test results, on techniques of structuring fault detection and diagnostic programs, and diagnostic data interpretation.

*Tests Derivation*—Methods of deriving tests for logic circuits have been extensively explored.[9,10,14,15] The objective is to generate a set of tests capable of detecting each member of a prescribed fault set. The most significant result that is applicable to circuits of practical size is the path sensitizing concept or the D-algorithm technique.[9–11] The idea is to assign a certain input test vector to a circuit so that faults along some path from input to output will cause the circuit output vector to be different from that obtained under the fault-free condition. For combinational logic, programs for deriving sensitized paths are fairly simple to implement. The running speed is also moderate for circuits with very few reconvergent fanouts. For sequential logic, there is no known technique that can efficiently handle circuits with even a moderate number of feedback paths. A practical approach, therefore, would be to design the processor organization with a minimum number of controllable feedback paths, as was suggested in an earlier section making the logic purely combinational for the purpose of testing. The path sensitizing techniques can then be used to derive a complete test set.

*Generation of Test Results*—The pros and cons of developing a digital fault simulator for generating test results as opposed to other alternatives (e.g., the manual method and the physical simulation approach) have been discussed by Manning and Chang.[16] It was concluded that the digital method is extremely useful in the early design stage to provide immediate feedback on the adequacy of hardware design and processor maintainability. The physical method seems to have an edge in computer time required to generate all the test results. However, it is not clear how the physical method can be used for a circuit realized with integrated circuit technology.

At present, for circuits with 100 logic gates a typical estimate of required computer time to generate test results is about one hour.[17] With improved techniques for fault simulation, the running can be substantially reduced so as to make the digital approach even more attractive.[16] Those readers who are interested in the detailed description of the development of a digital fault simulator can refer to several articles by Seshu. See References 19–21.

*Test Ordering and Minimization*—The test set and the test results generated through the simulation-process usually contain redundancy. In some real-time systems in which both program space and time are at a premium, it is desirable to select a minimum or near minimum set of tests that isolate faults only to the circuit package level. To accomplish this, the tests should first be arranged in "logical" order in the same manner as modules of the processor are ordered (see Figure 3(b)). This in effect constitutes a parallelism between the organization of the processor and the structure of testing procedure, which is considered to be a useful aid in isolating marginal and/or intermittent faults that produce inconsistent test results from diagnosis to diagnosis. Then, the test set for each module can be reduced by using one of the known methods for selecting an optimum set of diagnostic tests.[22–24]

*Program Structure*—The final phase in the design of a diagnostic testing procedure is to incorporate the tests and test results (obtained through the simulation process) into a diagnostic program. In order to minimize the overall program development effort (e.g., programming, debugging, integration and documentation) and to reduce the program maintenance effort (e.g., updating changes, etc.), the program structure should be modular, uniform and consistent. To accomplish these objectives, the use of the "data table" approach is recommended.

Basically, the program is composed of two parts: the control section and the data table section. The data table section consists of a sequence of standard entries, each of which specifies the operation of a particular test or test sequence for certain modules. Typically, each entry specifies (a) the input test vector(s) to be applied, (b) the prescribed length of time or number of central processor cycles the circuit is forced or allowed to operate, (c) the expected circuit response or output(s), (d) the information necessary to interpret the results, and (e) the required initialization information, if any. The information contained in the data table can be derived with the aid of a digital simulator. The control section, on the other hand, is a program designed to interpret these entries and perform functions such as initialization, segmentation of tests, interfacing with other programs, manipulation of test outputs, etc. Figure 7 illustrates the layout of a typical diagnostic program structure.

Experience on using this particular design approach reveals several advantages: (a) the design process becomes standardized, which in turn results in a large saving in program development; (b) programs are more easily modified, e.g., if the circuit changes, the majority of program alterations will be restricted to the data table section; (c) test results are easier to interpret, and (d) the control section can be written and debugged independently of the data table section. Also, the data table lends itself well to the participation of many designers, e.g., the register designers develop the data table for the registers, the decoder designers develop the data table for the decoders, etc. However, in systems where a large number of memory fetches can be penalized in time, it may suffer a slight drawback in that an increase in execution time of the program over the conventional approach may be realized. However, this problem is not serious as the diagnostic program is not a frequently executed program.

*Data Interpretation*—In large real-time systems, the diagnostic output usually corresponds to an enormous amount of data (e.g., for a processor of $10^4$ gates, a test vector might be represented by about 5,000 bits, where each bit designates the pass or fail of a test). In addition, the observed fact that test results of some faults are inconsistent from diagnosis to diagnosis demands a flexible data interpretation procedure.

Several techniques for resolving diagnostic data into faulty components or circuit packages have been described in the literature.[5] These techniques employ the concept of some form of fault dictionary in which each entry of the dictionary points to the set of faulty components or circuit packages producing the particular failure pattern(s). These patterns can be derived by simulation.[11,20,26]

The simplest form of dictionary is a listing of test results where a "0" indicates a test passed and "1" indicates a test failed. Faults are located by finding a match between the observed failure pattern and the entry listed in the dictionary. This technique is adequate to analyze failure patterns consisting of a relatively small number of failing tests. For fault conditions producing a large number of failing tests, a data compression technique to represent the pattern in some compact form (e.g., a fixed length number) is desirable in order to minimize the system repair time. The tradeoff between the isolation accuracy of the dictionary and the resolution provided by each of these techniques is discussed in Reference 26. The final choice of methods for interpreting diagnostic data for fault isolation depends on the allowable system downtime and the availability of skilled maintenance personnel.

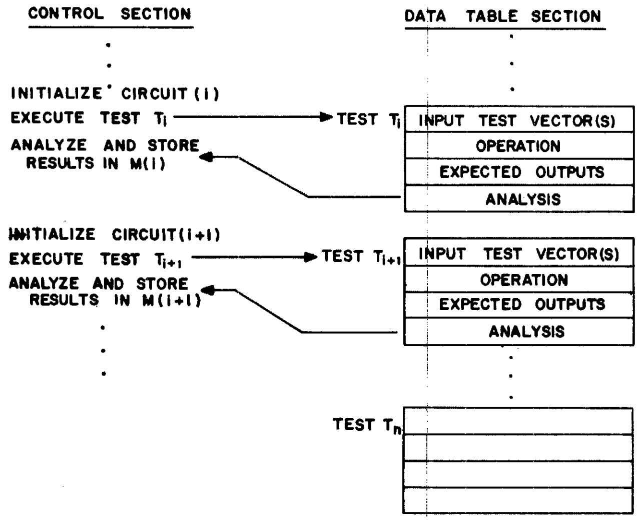


Figure 7—Layout of a diagnostic program structure

## CONCLUSION

In this article we have given a unified account of design principles for processor maintainability in real-time systems. The processor should be functionally well modularized with a minimum number of intermodular feedbacks. This is necessary to confine the effects of malfunctions as well as to facilitate programmed testing. To insure the validity of diagnostic data the amount of hardcore should be minimized, and ample test points must be provided to control the state of the machine, even under faulty conditions. Adequate system recovery mechanisms must also be incorporated to insure system sanity in a fault environment. Furthermore, the processor should have a clean interface with the external devices such as memory units and peripheral systems to enable the rapid identification of trouble symptoms to a subsystem level.

The design of processor logic circuits should be preceded by a thorough understanding of the failure modes of the circuit technology chosen for implementation. The elimination of circuit redundancy and the incorporation of a packaging scheme which provides good diagnostic resolvability are some other desirable prerequisites for a good maintenance scheme. Finally individual circuits should be examined to determine whether an all hardware, hardware-software, or all software maintenance facility should be provided.

The diagnostic program should be structured to efficiently implement the selected testing procedure (combinational or sequential). It should also provide a flexible operator interface to aid in isolating intermittent faults. Computerized fault simulation methods, which enable one to generate and evaluate the diagnostics, should be used throughout the design stages to provide adequate feedbacks on the effectiveness of system's diagnosability.

It is recommended that designers consider these guidelines in planning a new machine organization, designing logic circuits and maintenance tests so that an optimum mix of software and hardware for processor maintainability can be achieved. Because of the increased complexities of present and next generation computing systems, and because of the rapidly changing technologies, new maintenance techniques will have to evolve at an accelerated rate. We have only documented a few thoughts on guidelines for processor maintainability in real-time systems. Our opinions are obviously influenced by our training and experience. Since there are only a limited number of published documents on this subject, we encourage other workers in this field to present similar results.

## REFERENCES

1 W KEISTER et al
*No. 1 electronic switching system*
Bell System Tech Journal Vol 43 No 5 parts 1 and 2 Sept 1964

2 R H WILCOX W C MANN editors
*Redundancy techniques for computing systems*
Spartan Books 1962

3 W H PIERCE
*Failure-tolerant computer design*
Academic Press 1965

4 R E BARLOW F PROSCHAN
*Mathematical theory of reliability*
John Wiley and Sons Inc 1965

5 H Y CHANG W THOMIS
*Methods of interpreting diagnostic data for locating faults in digital machines*
Bell System Tech Journal Vol 46 No 2 Feb 1967 289-317

6 C V RAMAMOORTHY
*A structural theory of machine diagnosis*
Proc SJCC Vol 30 April 1967 743-756

7 R E STAEHLER
*No. 1 ESS service experiences—Hardware*
IEE Conf on Switching Techniques for Telecommunication Networks April 1969 463-466

8 A D FRIEDMAN
*Fault detection in redundant circuits*
IEEE Trans on Electronic Computers Vol 16 No 1 Feb 1967 99-100

9 D B ARMSTRONG
*On finding a nearly minimal set of fault detection tests for combinational logic nets*
IEEE Trans on Electronic Computers Vol 15 No 1 Feb 1966 66-73

10 J P ROTH
*Diagnosis of automata failures: A calculuc and a method*
IBM Journal of Research and Development Vol 10 1966 278-291

11 J P ROTH W G BOURICIUS P R SCHNEIDER
*Programmed algorithms to compute tests to detect and distinguish between failures in logic circuits*
IEEE Trans on Electronic Computers Vol 16 No 5 1967 567-579

12 W WORKMAN
*Failure modes of integrated circuits and their relationship to reliability*
Microelectronics and Reliability Vol 7 1968 257-264

13 J B KRUSKAL R E HART
*A geometric interpretation of diagnostic data from a digital machine*
Bell System Tech Journal Vol 45 Oct 1966 1299-1338
Based on a Study of the Morris, Ill Electronic Central Office

14 R D ELDRED
*Test routines based on symbolic logic statements*
Journal of ACM Vol 6 No 1 1969 33-36

15 J F POAGE
*Derivation of optimal tests to detect faults in combinational circuits*
Math Theory of Automata Polytechnic Press 1963 Brooklyn N Y 483-528

16 E G MANNING H Y CHANG
*A comparison of fault simulation methods for digital systems*
Digest of the First Annual IEEE Computer Conf 1967 10-13

17 E R JONES C H MAYS
*Automatic test generation methods for large scale integrated logic*
IEEE Journal of Solid State Circuits Vol 2 Dec 1967 221

18 E G MANNING H Y CHANG
*Functional techniques for efficient digital fault simulation*
Digest of IEEE Internat Conv March 1968 194

19 S SESHU D N FREEMAN
*The diagnosis of asynchronous sequential switching systems*
IRE Trans on EC Vol 11 No 4 Aug 1962 459-465

20 S SESHU
*On an improved diagnosis program*
IEEE Trans on EC Vol 14 No 1 1965 76-79

21 S SESHU
*The logic organizer and diagnosis programs*
Rpt R-226 Coordinated Science Lab Univ of Ill 1964 (AD605627).

22 R A JOHNSON
*An information theory approach to diagnosis*
Proc 6th Nat Symposium on Reliability and Quality Control 1960 102-109
23 H Y CHANG
*An algorithm for selecting an optium set of diagnostic tests*
IEEE Trans on EC Vol 14 No 5 1965 706-711
24 H Y CHANG
*A distinguishability criterion for selecting efficient diagnostic tests*
Proc SJCC Vol 32 1968 529-534
25 F J HACKL R W SHIRK
*An integrated approach to automated computer maintenance*
Conf Record on Switching Theory and Logical Design 1965 289-302
26 H Y CHANG
*Figures of merit for the diagnostics of a digital system*
IEEE Trans on Reliability Vol 17 No 3 Sept 1968 147-153

# Effects and detection of intermittent failures in digital systems

*by* M. BALL and F. HARDIE

*IBM Corporation*
Owego, New York

## INTRODUCTION

A great deal has been written during the past few years on the subject of diagnostic test procedures for digital systems. Almost without exception, however, the investigators have limited their interest to the detection and location of solid faults, and their test procedures are usually based on the assumption that either the fault exists for the running time of the test procedure or the time interval between the fault occurrence is less than the required time to run the test.

In practice, experience has shown that field failures in digital systems used for aerospace application (e.g., Titan and Saturn vehicle guidance computers) tend to be intermittent in nature. The authors believe that this experience is testimony to the efficiency of the current diagnostic test procedures in screening solid faults from digital systems before delivery for field use, not that failures which develop in the field tend to be intermittent. That is, diagnostic testing of aerospace digital systems using the advanced test procedures available today generally detects all solid faults but only a small portion of the intermittent faults that exist in any digital system prior to delivery to the field. The residue of intermittents in the system which escaped detection eventually make their presence known during field operation.

The reason for the emphasis on diagnosis of solid faults is the relative complexity involved in the diagnosis of intermittent faults. This is the natural course of evolution in system design as well as in biology-adaption to basic environmental requirements with later complex specialization.

In an attempt to direct the evolution of diagnostic techniques along the channels leading to efficient detection and location of intermittent failures in digital systems, the authors conducted a series of experiments on the effects and detection of intermittent failures. Over 500 hours of IBM 7090 time were accumulated using a sophisticated logic simulator to evaluate the Saturn V Launch Vehicle aerospace computer operation in both normal and failure modes. The purpose of these experiments was to determine the effects of intermittent failures on computer operation rather than to investigate the mechanisms of failure, and to evaluate the detectability of classes of failure rather than to develop specific techniques for failure detection.

In this study solid faults were treated as a special case of the general class of intermittents. That is, a solid fault was treated as an intermittent whose duration exceeds the running time of the test program. The simulated intermittents were made to vary in duration from 500 nanoseconds upwards (one clock time of the simulated computer), and were specified in the computer logic at randomly chosen points of combinational and sequential circuits. A total of 792,884 intermittent failures were simulated to give a realistic statistical sample. These intermittent points were chosen to occur in the program control and arithmetic sections of the simulated computer.

For each intermittent a record was kept of the time of error occurrence, time of error detection and the number of failures which caused a difference in operation from a "good" machine. From these records the probability of detection was calculated assuming a

TABLE I—Intermittent detection capabilities

| Unit | Total Failures | Failure Duration | Failures Causing Incorrect Operation | % Affected Logic | Failures Detected | Detected % |
|---|---|---|---|---|---|---|
| Adder/ Subtractor | 267,894 | 500 nanosec | 22,276 | 8.4 | 1,113 | 5 |
| Multi/ Divide | 269,590 | 500 nanosec to 5 millisec | 22,376 | 8.3 | 1,122 | 5 |
| Program Control | 255,400 | 500 nanosec to 5 millisec | 44,704 | 17 | 252 | 0.5 |

"perfect" error detector. The results showed that many intermittent failures exert only a weak influence on the correct operation of synchronous logic circuits. As shown in Table I, approximately eight percent of the simulated failures caused the arithmetic element to perform incorrectly, with a comparable (five percent) probability of detection by the "perfect" error detector.

## *The system simulator*

One of the most serious problems confronting the designers of digital systems is the task of verifying proposed design features. Both manual analysis and simulation techniques are used to aid in this task. During the design and development phase of the Saturn V Launch Vehicle Digital Computer, a Fault System Simulator was developed* by IBM to provide the means of (1) verifying the logical integrity of the digital equipment, (2) evaluate design changes before commitment to hardware, and (3) evaluating test programs. During the course of its use, however, emphasis gradually shifted to a special simulator application which generate information on the characteristics of machine operation to aid the engineer in diagnosing malfunction symptoms. One of the most significant series of simulator experiments was concerned with evaluating the sensitivity of the digital logic to intermittents.

The system simulator consisted of a compiler, failure injector, logic simulator, and evaluation programs. These programs operated on the IBM 7090 computer as shown in Figure 1. The compiler program produced 7090 instructions for the logic portion of the simulator program. The failure injection program allowed the introduction of selected faults into the logic portion of the simulator program on the component level—that is, open or shorted diodes and transistor outputs

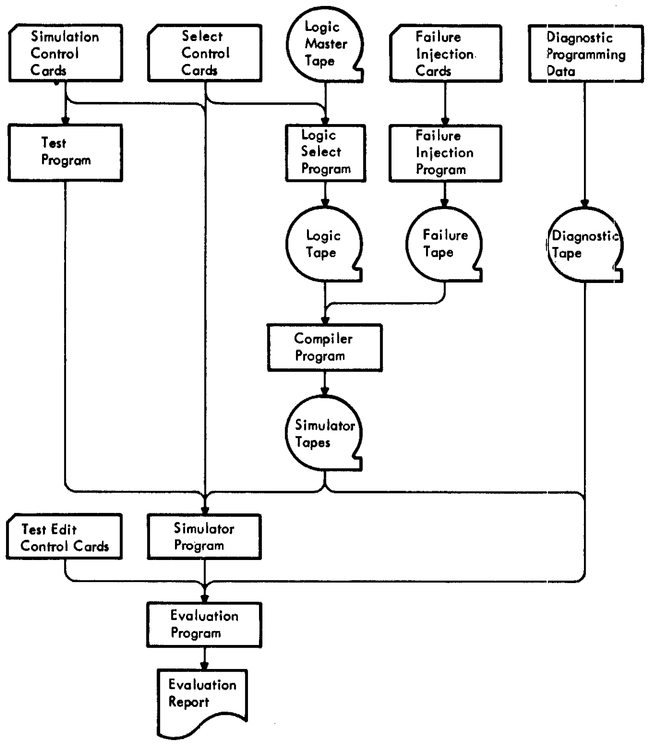


Figure 1—Simulator flow diagram

* Design and Use of Fault Simulation for Saturn Computer Design, by F. Hardie & R. J. Suhocki—IEEE Trans. on Electronic Computers Vol EC-16, No. 4 August 1967 p. 412-29.

stuck to a logical zero or a logical one. The simulator program operates on a 7090 description of the digital equipment (a logic master tape) to simulate the logical behavior of the equipment in normal operation and in various failure environments.

The simulator program executed special test programs and displayed, by means of print-outs, the state of selected logic nodes or register contents at every clock time of an instruction cycle. In investigating the behavior of equipment containing logic failures, simultaneous failure environments were provided by using parallel simulation techniques, and the system states for each environment were determined simultaneously. Of the 36-bit 7090 word, 3 bits were used to represent the normal system state and each of the remaining 33 bits were used to represent a failed state Multiple faults were simulated by injecting 2 to 25 failures into a single bit position.

Up to 100 logic test nodes were available for printout in each normal or failure environment. Special pseudo operation codes allowed additional nodes to be retrieved as required. Another pseudo operation code caused the contents of selected registers to be placed on the simulator output tape for use by the evaluator program.

The evaluator programs identified fault symptoms and correlated these symptoms with the injected failures. The output of the evaluator was a report of detected errors, undetected errors, accuracy of diagnosis, and general behavior of the digital equipment.

*Simulator applications*

The primary applications of the system simulator can be grouped into four general categories: design evaluation, failure evaluation, data generation, and data analysis. The obvious use of the simulator was to provide early and rapid verification of the logical integrity of the basic hardware designs of digital equipment. In addition to checking the basic logic, the simulator was used to determine whether certain design ground rules were satisfied by the circuit designs, and even whether the ground rules themselves were adequate. For example, individual circuits were checked against fan-in and fan-out constraints. In addition, the constraints themselves were checked against drive and load requirements by applying random and worst case parameter values to the drives, driven circuits, and circuit loads.

Delay simulation, incorporating logical element delay characteristics in the logic simulator, was used to analyze the nature of digital signal propagation in the computer designs. Several race conditions were detected by the delay simulations which were corrected by modifying equipment initialization procedures or by design changes.

Operational and test programs were evaluated on the system simulator. Although functional program simulators provide nearly error-free programs from the standpoint of information flow, an appreciable amount of program debugging is usually required when the program is first used with the hardware. Logic simulator evaluation of programs reduced this final debugging phase to a minimum.

The applications discussed so far pertain to properly operating equipments. The logic simulator should be regarded in such applications as a tool to aid in design analysis and not as a replacement of manual analysis and engineering judgment. In the area of failure mode analysis, however, the simulator as a tool becomes even more important because of the inherent difficulty in determining the behavior of failed machines, and especially in identifying the fault from the failure symptoms.

The failure injection program and diagnostic evaluation programs provide a failure evaluation capability for the system simulator. Test programs for equipments were evaluated for their failure detection and fault isolation capabilities. Built-in test circuitry and test point configurations were evaluated in the same manner. Optimum placement of detection circuits and test points was determined by successive simulations.

Although the evaluation applications represent perhaps the most important use of the system simulator, the simulator also possesses a capability of generating data which is useful not only in design and test of the system but also in increasing the capability of the simulator itself. For example, a diagnostic catalog can be generated as a by-product of a test program evaluation which relates each injected fault to the resulting failure symptoms. The catalog is then available for use in evaluating diagnostic programs or procedures in further simulations.

One of the applications of the logic simulator which is generally very difficult to perform manually is to trace the propagation of an error caused by a component failure, especially when the failure produces a loss of program control. Such traces can be generated by logic simulation, however, and have important diagnostic value in identifying system faults. The status of the failed equipment at every clock time can be determined by monitoring over a hundred nodes or test points internal to the equipment logic, as well as the equipment interface. A summary of simulator applications is shown in Figure 2.

- Design Evaluation
    - Hardware
        - Basic Logic
        - Design Ground Rules
        - Delay Simulation
    - Software
        - Operational Programs
        - Test Programs
    - Design Changes
- Failure Evaluation
    - Test Programs
        - Error Detection
        - Efficiency
        - Diagnostic Capabilities
    - Circuit Sensitivity
        - Error Propagation
        - Failure Effects
- Data Generation
    - Node Data
    - Error Traces
    - Diagnostic Catalog
- Data Analysis
    - Laboratory Support
    - Field Failure Analysis

Figure 2—Simulator applications

*Simulation of intermittent failures*

The application of the simulator which is the primary concern of this paper was a series of experiments to determine the sensitivity of logic to intermittent failures. Intermittents simulated by the failure injection program were made to vary from one clock time to the cycle time of the test program (representing a solid failure). These faults were injected at randomly chosen points in the equipment logic and at random points in the test program.

For each intermittent a record was kept of the time of occurrence, time of detection, and the number of failures which caused a difference from the "good" machine. The results of the simulation indicated that many intermittent logic failures had very little effect on the operation of the digital equipment—less than ten percent of the total failures injected into the simulator program caused the logic to perform incorrectly. Analysis of the simulation results disclosed that this masking of failures by the logic was due primarily to

- the extensive use of combinational logic
- the clocking of the AND gates which feed and/or gate the logic levels from the sequential circuits.
- the duration and frequency of the intermittent failure.

These simulation results and conclusions were based on a relatively small statistical sample—a few hundred simulated failures. In order to obtain a realistic statistical sample, the failure injection program was modified to execute the following procedure:

1. The logic failure was initiated at the first clock time of the test program.
2. The test program was executed until a state difference was detected by the simulator program between the logic under examination and a "good logic" reference.
3. Upon failure detection, the time of detection and failure symptoms were recorded, the logic under examination reset to the same state as the reference logic, and the test program advanced to the next clock time.
4. The procedure was repeated for one full cycle of the test program.

The immediate data from this simulation provided a measure of the sensitivity of the logic to intermittent failures of one clock time duration. That is, the portions of the test program during which the injected faults cause a deviation from normal operation were identified. The same data was used to provide a measure of the sensitivity of the logic to intermittent failures of longer durations than one clock time by manipulating the data with simple editing programs rather than by further simulation, making it feasible to accumulate information on an equivalent of over a half million simulated failures.

To assure the validity of the above techniques, the quantitative results concerning the sensitivity of the logic to intermittents obtained by the first method of actually simulating failure durations of one clock period and then manipulating the data with special edit programs, were compared and found to be closely correlated. The combined data from both simulation experiments was then used to derive a series of curves representing the sensitivity of the logic to intermittents of various durations, two of which are shown in Figures 3 and 4. The ordinate in each figure is the probability that the intermittent will cause a malfunction in logic

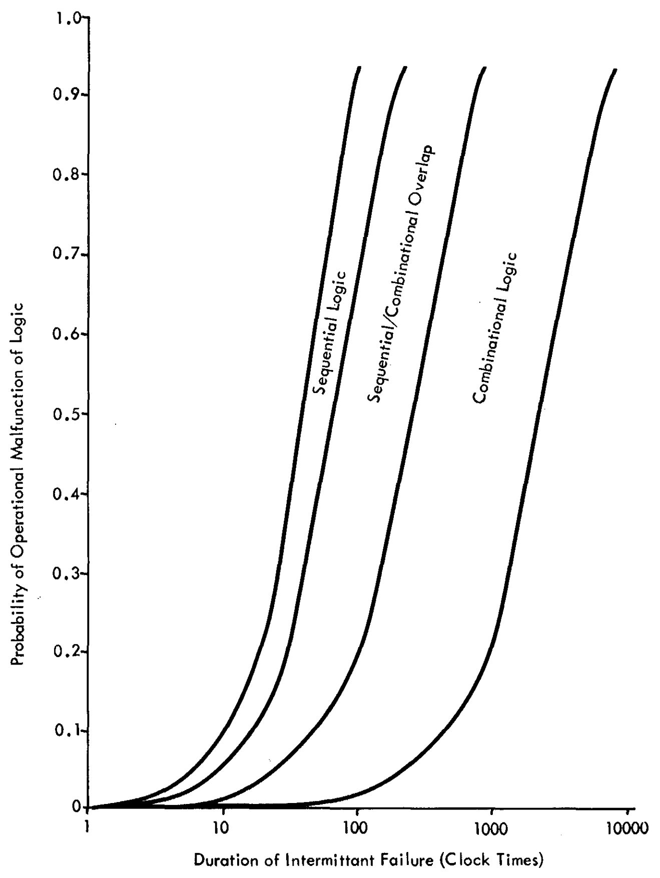


Figure 3—Sensitivity of arithmetic logic

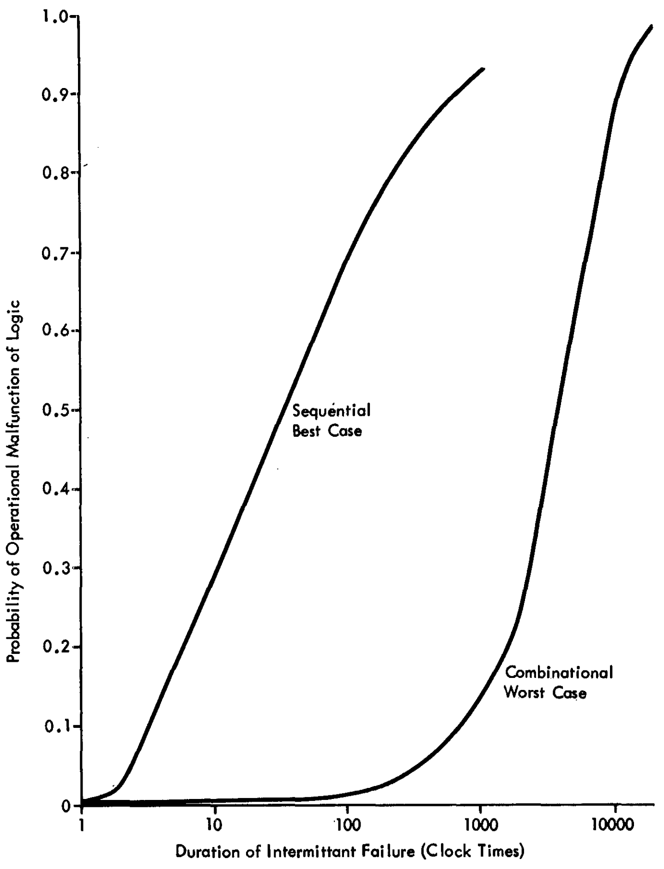


Figure 4—Sensitivity of multiply/divide logic

operation, while the abscissa is the duration of the intermittent.

The sensitivity of the logic was found to vary appreciably not only with the class of logic (combinational or sequential) but with the operational function of the logic circuitry as well. This condition necessitated the plotting of sensitivity versus fault duration individually for different areas in order to obtain meaningful relationships.

A summary of these results is given below:

- There is a smaller probability of detecting intermittent failures in combinational (AND-OR) circuits than in sequential (LATCH) circuits.
- There is a very low probability of detecting a single occurrence intermittent failure on a logic page (average population of 120 AND, OR, invert type circuits). This condition exists because many intermittents do not make the "failed" logic act different from the "good" logic and the detection of intermittents requires that the logic must be exercised by appropriate data for the failure to be detected.
- For these injected intermittents, a fault existing for one clock time was virtually undetectable; one existing for ten computer word times was about 50 percent detectable; and one existing for 50 computer word times was almost 100 percent likely to be detected.
- There is a wide variation of error detection sensitivities between computer modules.

*Test program efficiency*

An analysis of simulation results was performed to determine the quantity and type of information which should be generated by a test program to assure a reasonable probability of error detection and fault location in the digital equipments. Figure 5 shows the efficiency of the test program versus the size of the test program for various types of failures. Curve a represents a solid failure. Curves b and c represent an intermittent failure of 100 clock time duration in typical

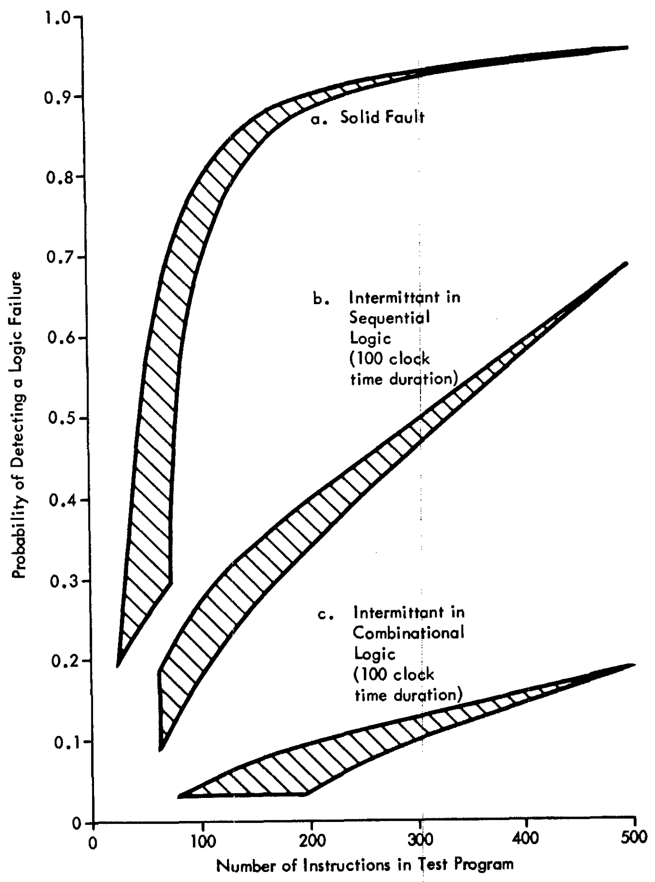


Figure 5—Efficiency of test program vs. program size

sequential and combinational logic, respectively. Note that, although a reasonably high efficiency of detecting a solid failure was achieved with a relatively short test program (90 percent with 200 instructions), the probability of detecting the intermittent was almost linear with program size.

Many different types of error symptoms were produced as a by-product of the simulation experiments. Each symptom was analyzed to determine its individual and combined value in identifying logic failures. Figure 6 is a summary of the results of this analysis for solid faults. The relative diagnostic values of the error symptoms in identifying intermittent failures are about the same except that the percentages will be less according to the duration of the intermittent.

Due partly to unavoidable redundancy in a test program (by which a logic element is exercised more than once) and due to error propagation in digital systems, an error in logic operation resulting from a failure of a logic element can occur several times during the execution of the test program. The first line of Figure 6 indicates the program instruction during which an error was first detected. The second line indicates the phase, bit and clock time that the error was first detected. The third line indicates the first three program instructions during which an error was detected. The remaining lines indicate various combinations of the above test parameters.

| *Observed Failure Symptom or Parameter* | *Failures Identified* |
|---|---|
| First Program Step of Detected Error | 10.5% |
| Time of First Detected Error | 28.1 |
| First Three Program Steps of Detected Error | 63.2 |
| First Program Step of Detected Error and Time of First Detected Error | 71.8 |
| First Three Program Steps of Detected Errors and Time of First Detected Error | 83.2 |
| First Three Program Steps of Detected Errors and Time of Each Detected Error | 97.5 |

Figure 6—Symptom failure correlation

## CONCLUSIONS

The series of simulation experiments described above strengthened the authors' opinion that the prevalence of intermittent failures of digital equipments in the field is due to the relatively low efficiency of current test techniques in screening such failures before delivery of the equipment to the field. That is, although current test techniques cause most of the solid faults which are "built into" the equipment during fabrication to be discovered before release to the field, a large residue of intermittents slip through the test screen and cause operational errors during field use.

The simulation experiments described above did very little in the way of deriving a solution to the problem of intermittent failures. No attempt was made to determine the mechanisms or characteristics of actual intermittent faults in existing digital equipment. The experiments were designed only to examine the sensitivity of digital logic to intermittent faults in general, without regard to mechanisms of failure.

The simulation results indicated to the authors that current test techniques, slanted toward detection and location of solid faults in digital equipment, are adequate for solving the problem of intermittents. The experiments showed a rather surprising insensitivity to intermittents of short duration. Although this insensitivity may seem to be a fortunate characteristic for actual operation, it makes the problem of testing infinitely more difficult.

Two general approaches to the test problem are obvious:

- develop better test techniques for detecting and locating intermittent faults, and
- develop techniques for making the intermittents appear solid.

The second approach has found widespread acceptance, as indicated by the common use of vibrational and thermal stimuli to force intermittent faults to expose themselves during factory checkout. In this way many intermittent faults are detected that may otherwise have slipped through the factory test screen. The prevalence of intermittent failures during field operation, however, testifies to the inadequacy of this approach by itself.

A third approach is, of course, to design the equipment to be absolutely insensitive to intermittent logic failures. Instruction retry, check point rollback and redundancy are being advanced as possible solutions. Redundancy, especially triplicated logic with voting,* has proven very effective in this area, but not without cost in hardware and power. Eventually, when logic hardware becomes sufficiently inexpensive, redundancy may very well be the way of life and the intermittent problem will have been solved.** Meantime, there remains urgent need for developing better test techniques for detecting and locating intermittent faults in digital equipment.

The greater part of test and maintenance cost of computer systems today is spent on detecting and isolating intermittent failures. Intermittents have comprised over thirty percent of pre-delivery failures and almost ninety percent of field failures in several computer systems known to the authors, and this seems to be the trend in present computer technology. Unfortunately, most of the current research in diagnostic techniques is concerned with the detection and location of solid failures.

Logic simulation has provided a powerful tool for studying the effects of intermittents in specific computer organizations, but in itself is not a solution to the cost problem. Even when these effects have been identified, the techniques for designing a computer to be intermittent-resistant or for testing a computer to locate intermittent failures are not yet state-of-art.

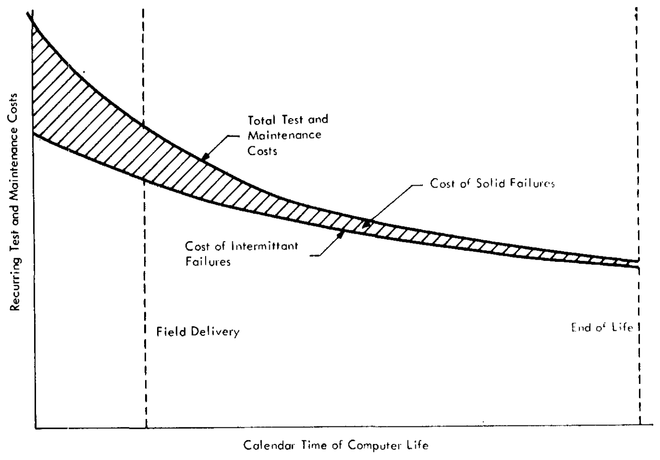


Figure 7—Cost of failures

Figure 7 shows a typical curve of the relationship of the costs of testing and maintaining a computer system from its initial assembly to the end of its useful life. The following conclusions may be evident from the figure:

- Intermittent failures are far more costly in test and maintenance than solid failures.
- The cost ratio of intermittent to solid failures increases with system usage, especially following delivery to the field. The reason for this trend is probably the better screening of solid failures by current test techniques.
- The cost of field maintenance remains high with usage, and most of the cost is due to intermittent failures. This large residue of intermittent faults is probably due to inefficient test screening rather than to new faults.
- The costs of a computer system tend to be monotonically decreasing with use. End-of-life is forced by obsolescence rather than by wear-out.

* IBM Proposes Triple-Redundant Computer, by M. Ball and F. Hardie, Computer Design Vol. 6, pages 34-36, Nov. 1967.
** Self-Repair in a TMR Computer by M. Ball and F. Hardie. Computer Design Vol. 8, No. 4, pages 54-57, April 1969.

# Modular computer architecture strategy for long term missions

*by* F. D. ERWIN

*Hughes Aircraft Company*
Fullerton, California

and

E. BERSOFF

*NASA Electronics Research Center*
Cambridge, Massachusetts

## INTRODUCTION

Long term mission reliability of a modular computer has been studied at Hughes Aircraft Company as a consequence of a study with NASA ERC.[1,2] Particular interest lay in the attainment of long term reliability with modular computer organization and the effects on reliability of variations in modular organization. The results of this investigation are presented in this paper.

In the past, the designers of aerospace computers have concentrated on increasing computational speed and arithmetic capability within stringent weight and power limitations. There seems to be little doubt that aerospace computers will soon be extremely fast, versatile and compact. A requirement for long term system reliability has been developing and may drastically change the nature of the on-board computer. Extremely long missions are being planned which require a computer to operate for one to five or more years after launch. Current on-board computer systems are not adequate for this task.

One promising approach for achieving reliability and flexibility is through modular design, where independent physical modules, functionally organized (e.g, memory, arithmetic, control, Input/Output) can be added or deleted to adapt to the required performance and processing needs in terms of speed and reliability. Improvement in reliability through the use of additional hardware has been receiving growing attention in the aerospace computer community.[3]

Specifically in this paper, a technique will be described which when properly applied will determine a computer configuration which can satisfy a required probability of mission success for some stated mission duration. It is assumed that some basic computer system exists which can perform the required computations; what remains is to determine which additional computers or sub-computers should be added to provide the necessary system reliability.

### *A modular computer design*

Several techniques exist which are designed to increase the reliability of any given computer system. The approach taken here is to have a single computer perform all mission computations while a number of other computers remain in a dormant mode until the working computer fails. At that time, the failed computer is turned off and one of the dormant computers is turned on to resume the computations. The size, weight, and power restrictions will typically limit the number of spare computers that are available. An additional refinement to this concept is to segment

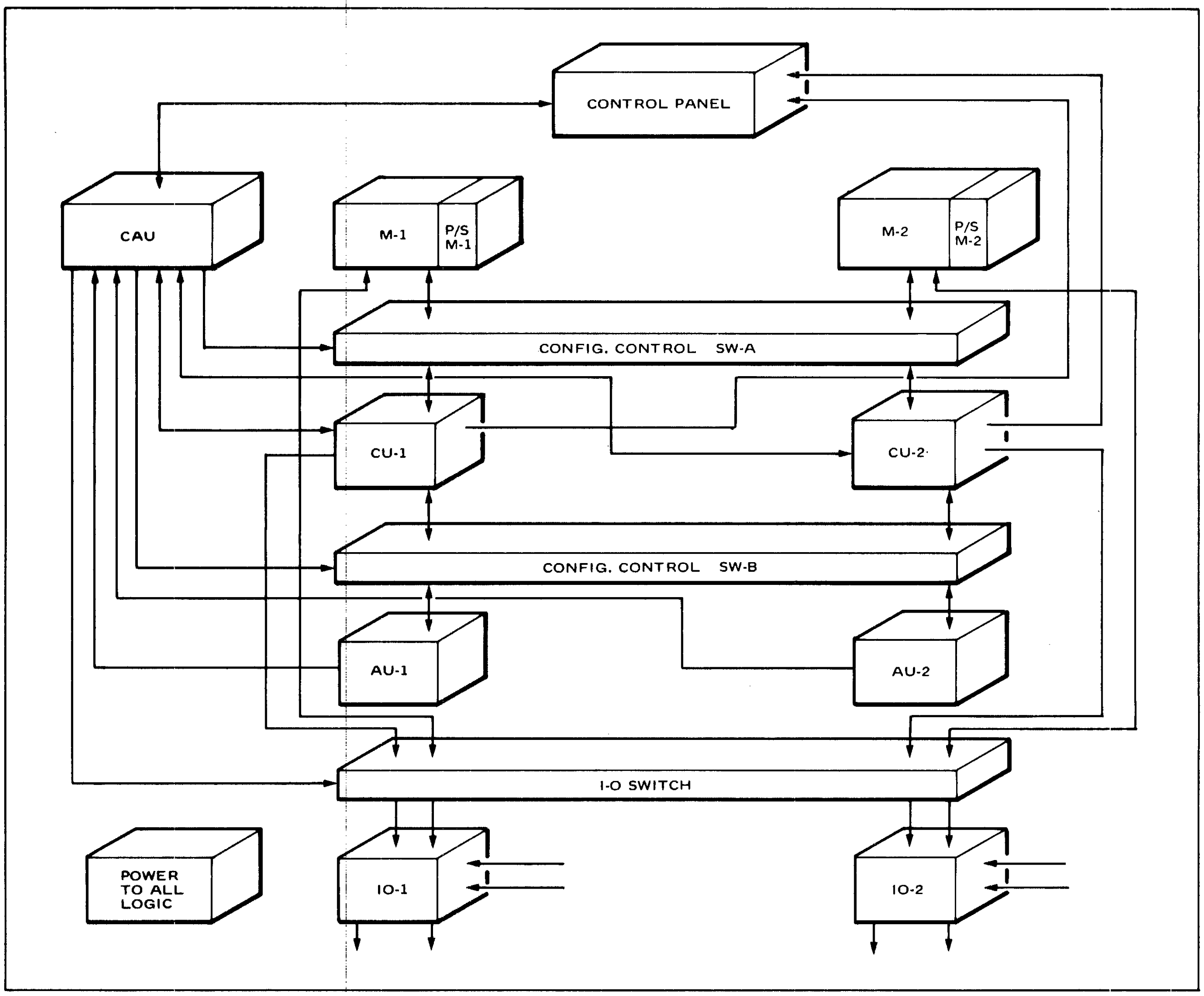


Figure 1—Modular computer breadboard

each computer into functional modules. One possible method is to isolate the memory, central processor, and I/O functions as in the Hughes H4400 computer. Another is to partition the computer into discrete memory, control, arithmetic, and I/O units, as in the NASA modular computer. This second approach will increase the total system parts count, but the increased modularity may ultimately enhance system reliability.

A mathematical model was constructed which permits the evaluation of computer reliability for the various configurations.

The two configurations mentioned above were normalized in terms of logic complexity to the NASA modular computer and the mission reliability was evaluated. Analysis showed that short term reliability using voting techniques as would be done during boost phase, favors the H4400 modularity concepts whereas the long term reliability tends to favor the NASA modular computer partitioning. In either case, the long term reliability is very sensitive to the logic distribution within the modules and to the basic reliability of the components.

A breadboard modular computer of the MCB with two modules of each type (a two column system) is being constructed. These modules are of sufficient complexity to prove many of the points under consideration. First, the system may be configured in a one active, one standby fashion so that the techniques of error detection and reconfiguration may be explored. Figure 1 is a block diagram of the NASA computer breadboard. The CAU (Configuration Assignment Unit) is the module that provides for continuation of system functioning under module or switch failure. It controls the activation and connection of standby modules into the operating system and failed modules out of it. The system can be configured so that one active string (memory, control, arithmetic, I/O) performs computations while the other unused modules remain in a standby state. If a failure occurs in any module, the module can be turned off and a standby module switched in. With the breadboard, if more than one of any distinct module fails, the system fails.

In order to compute the reliability of the modular computer, it is necessary to know the number of components in each module and their failure rates. Since it is also interesting to examine the case of the three modules (memory, central processor, I/O) configuration, Table I presents a breakdown of components for each case. As is shown in the figure, the basic component can be either the gate or the integrated circuit.

Enough experience with the NASA breadboard has been accumulated to instill a high level of confidence that the modular computer concept is sound and indeed workable. The system may be arranged in a TMR fashion during the boost phase of a mission when calculations are proceeding too rapidly to allow reconfiguration. After boost, two of the three computers would be turned off and the system would enter the one-active two standby mode. A block diagram of the proposed system appears in Figure 2. With the exception of the CAU which would be more complex due to the additional modules it must service, the component counts for each of other modules should be approximately the same as listed in Table I for the two column system. It will be shown that under some conditions even dual redundancy for the working modules would not provide adequate reliability for a five year mission. Several things can be done to eliminate this deficiency. First, and most fundamentally, the components (gates, IC's, or LSI's), can be made more reliable. Second, different configurations can be structured such as 4 column and others so as to enhance the mission reliability. Certainly, the nature of the mission and weight constraints will impose a limit on hardware launched. The control unit (or alternatively the central processor) is clearly the largest single module of the system. If only an additional control unit were carried, total reliability would be increased. It will be shown, however, for a five year mission, considerable reliability enhancement must be made to the CAU as well as improvements to the other modules.

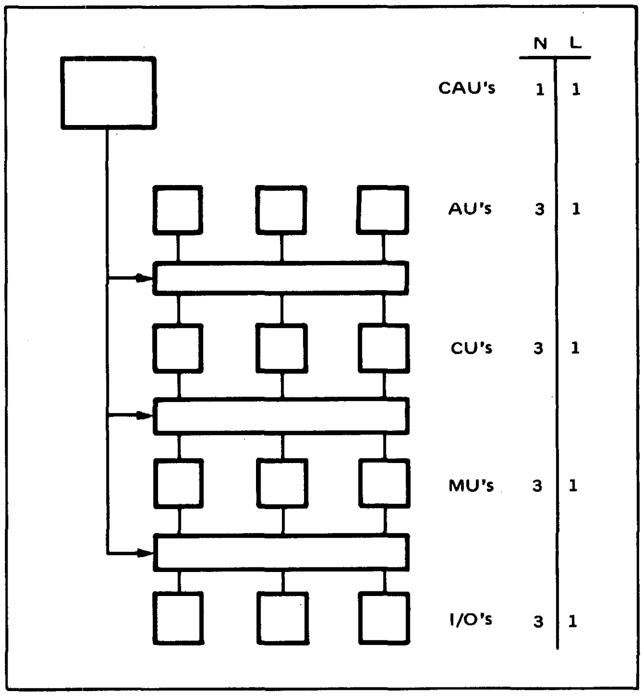


Figure 2—Three column modular computer

Several techniques designed to improve the CAU are required if the computer is to function for several years. One is to build the CAU with as few components as is possible; assuming equal component reliability the fewer components it has, the longer it will last. A further increase in reliability can be obtained by triplicating the CAU and using TMR voting on its output. Although a TMR configuration ultimately becomes less reliable than a single unit, this does not occur until about .7 times the mean life of the individual unit. Since the voter unit has relatively few components, its mean life is very long and therefore TMR is to advantage here. The type of system discussed and illustrated in the figures is known as a closed system in that at the beginning all equipment is present with no additions or repairs possible thereafter.

In analyzing system reliability, two important points for consideration are the rate of degradation of standby units and the switch reliability.

Kletsky[5] shows that the mean life of a closed modular set cannot be increased significantly when active and standby failure rates are assumed equal. Therefore, it is necessary to obtain a value for this ratio. Little direct data, however, is available for "d", the ratio of standby failure rate to active failure rate. However, data reported by Nerber[6] indicate that d is considerably less than unity. Nerber analyzed field data for over 100 transistorized guidance computers. From this data a maximum value of d can be inferred to be about 0.33. A more recent analysis of Minuteman II computer failures by Watson[7] shows that the expected value of d for integrated circuits is 0.55. A lower bound for this ratio appears to be 0.12. Extrapolation indicates that the ratio will decrease as more data is gathered. For future missions, it seems conservative to assume that the ratio will lie below 0.5 with 0.1 reasonably attainable. Though this ratio may further decrease as more is learned, the greatest significance for the closed module set reliability is effected with a d of the order of 0.1.

The reliability model must also consider the effect of the switches. Though the switch size is held at a minimum (typically 8 data bits + 5 control) its effect upon long term reliability if not properly treated can be great. For instance, the reliability of a single crosspoint (with a normalized failure rate, $\lambda_S = 2 \times 10^{-7}$) is .991, for a $3 \times 3$ switch .925, and the probability that 3 crosspoints out of the $3 \times 3$ switch will be operational is $1\text{-}10^{-10}$. Thus, for long term system reliability of the order of .99 proper treatment of the switch reliability is imperative.

The following paragraphs describe the mathematical model developed to accommodate the above factors

*Reliability model*

The block diagram of Figure 2 will serve as a framework for the mathematical model. The mathematical model is somewhat more general than Figure 2 in that there may be more than four levels (a level consists of all modules of one type) of modules with the j-th level having $N_j$ modules out of which $L_j$ are required operational with $N_j$-$L_j$ being in standby at the beginning. The computer system is operational as long as $L_j$ modules are operational at the j-th level for all j's with unfailed switch capability for interconnection of the $L_j$'s from level to level for all levels and the CAU is operational.

The switch between each level allows connection of any module at one level to any module at the other level. Further, the switch is designed so that independent failures may occur in the switch such that certain switching connections are disabled without affecting other connections (all other type switch failures can be associated with a module). Thus, one failure in a switch may make connection of module 1 of one level to module 2 of another level impossible without affecting the connectability of module 1 to module 1, module 2 to module 2, or module 2 to module 1. Figure 3 illustrates a typical switch.

It is desired to find the reliability of the total modular computer system at any instance of time. L modules must operate at each level as well as the switches to interconnect them. The problem is approached by first finding the reliability at each level then iterating from level to level including at each step the switch reliability.

If switching redundancy is applied to a module level with L replicas operating and N-L in standby the reliability according to Kletsky (Reference 5) can be given as an inverse La place transform.

$$R_R = 1 - \mathcal{L}^{-1}\left\{\frac{1}{S}\prod_{K=R-L}^{N-L}\frac{(L + Kd)\lambda}{S + (L + Kd)\lambda}\right\} \quad (1)$$

where R is the number of modules operational (unfailed) for $L \leq R \leq N$ and $\lambda$ is the active failure rate of the module while d is the ratio of standby to active failure rates.

The Reliability can be reduced to:

$$R_R(t) = \sum_{K=R-L}^{N-L}\left\{\epsilon^{-(L+Kd)\lambda t}\left(\prod_{\substack{h=R-L\\h\neq K}}^{N-L}\frac{L/d + h}{h - K}\right)\right\} \quad (2)$$

Since failure is characterized by independent random variables the probability ($P_R(t)$ ) of exactly N–R+1 failures (R operational) is equal to:

$$P_R(t) = R_R(t) - R_{R+1}(t)$$

taking the difference and combining terms leads to:

$$P_R(t) = \sum_{\substack{P=K=R-L\\P\neq R-L}}^{N-L}\left(\frac{(L/d + P)}{R\text{–}L\text{–}P}\right)\cdot \epsilon^{-(L+Kd)\lambda t}\prod_{\substack{h=R-L+1\\h\neq K}}^{N-L}\left(\frac{L/d + h}{h - K}\right) \quad (3)$$

To model the effect of the switches a recursive procedure is used beginning at Level 1, then Level 2 and onto the highest level.

Let $P^{(j)}_{RS}(t)$ designate the probability that at time t exactly R modules are operational at the j-th stage and exactly S ($S \leq R$) of them may be reached from Level 1 through a path of operational modules and subswitches at the lower levels (Reference 8). Since R N the states of the process that yield at least one computer's worth of capability (for the higher level) correspond to:

$$(R, S) = (1, 1), (2, 1), (2, 2), (3, 1), (3, 2), (3, 3), \cdots, (N, 1), (N, 2), (N, 3), \cdots, (N, N)$$

with reference to the switch of Figure 3, the equation governing transition between the various levels is given by:

$$P^{(j)}_{RS}(t) = P_R^{(j)}(t) \binom{R}{S} \sum P^{(j-1)}_{UV}(t) \cdot W^{(j-1)}_{RSV} \tag{4}$$

$$L_{j-1} \leq V \leq U \leq N_{j-1}$$

where $W^{(j-1)}_{RSV}$ is the probability that exactly S modules at level j are connectable through the switch from V modules at level j — 1. For exponential failure:

$$W^{(j-1)}_{RSV} = (1 -(1 -e^{-\lambda'(j-1)^t})^V)^s(1 -e^{-\lambda'(j-1)^t})^{V(R-S)} \tag{5}$$

where $\lambda'(_{j-1})$ is the cross point failure rate of the switch at the j–1st level.

Substituting Equations (3) and (5) into Equation (4) yields the expression for the probability

$$P^{(i)}_{RS}(t) = \sum_{\substack{p = K = R - L_{(j)} \\ p \neq R - L_{(j)}}}^{N_j - L_j} \left\{ \frac{L_{(j)}/d + p}{R - L_{(j)} - p} \right.$$

$$\left. e^{-(L_{(j)}+Kd)(j)^t} \prod_{\substack{h=R-L_{(j)}+1 \\ h \neq K}}^{N(j)-L(j)} \frac{L/d + h}{h - K} \right\} \cdot \tag{6}$$

$$\left\{ \begin{array}{l} \binom{R}{S} \sum P^{(j-1)}_{UV} (1 -(1 -e^{-\lambda'(j-1)^t})^V)^S . \\ L_{(j-1)} \leq V \leq U \leq N_{(j-1)} \end{array} \right.$$

$$\left. (1 -e^{-\lambda'(j-1)^t})^{V_{(R-S)}} \right\}$$

where the number of modules $N_{(j)}$, those required operation simultaneously, $L_{(j)}$, and a module failure rate $r_{(j)}$ are variable from level to level.

Thus, the $P^{(i)}$ (t) vectors are obtained recursively starting first from $P^{(1)}$ (t) which is given componentwise by:

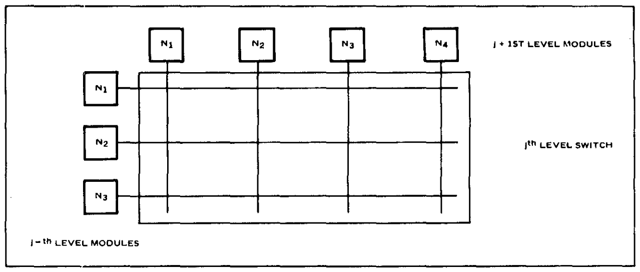


Figure 3—Module-switch-module relationships

$$P^{(1)}_{RS}(t) = \sum_{\substack{p=K=R-L_1 \\ p \neq R-L_1}}^{N_1-L_1} \frac{(L/d + p)}{R - L - p}$$

$$e^{-(L_1+Kd)\lambda_1 t} \prod_{\substack{h=R-L_1+1 \\ h \neq K}}^{N_1-L_1} \frac{L_1/d + h}{h - K} \quad \text{if } S = R \tag{7}$$

$$= 0 \qquad \text{if } S \neq (R$$

The total system reliability then including modules, switches, and CAU is given by:

$$R(t) = R_{CAU} \prod P^{(m)}_{RS}(t) \qquad L_m \leq S \leq R \leq N_m \tag{8}$$

where m is the highest module level and the reliability of the CAU is given by $R_{CAU}$.

R(t) then gives the probability that at time, t, at least one string of modules ($L_j$ at each level) is connectable and operational and further that the CAU can switch modules in and out of the operating string when needed.

Two assumptions implicit in this derivation should be noted. First that in assuming a strict exponential failure distribution for the modules wherein a module was assumed to fail at a rate λ if in the active state and λd if in the passive state, no allowance was made for failures which might occur by change of state transitions. No algorithm exists for switching modules on or off to form an operational string which guarantees full utilization of the remaining system without requiring that some modules be placed in successive modes of active and standby states. It is therefore assumed that module failure is not influenced by its history of active and standby state transition. To solve the problem otherwise becomes extremely difficult.

The second assumption is that all crosspoints of a switch are assumed to be in the active state whether or not the corresponding modules are presently connected, and therefore the failure rate of crosspoints fail only at the active rate (no standby rate).

The numerical calculation may be facilitated by rewriting Equation (3) as:

$$P_R(T) = \binom{L/d + N - L}{N - R}^{*} \sum_{K=R-L}^{N-D} (-1)^{-R+L+K} \binom{N-R}{N-L-K} e^{-(L+Kd)T} \quad (9)$$

Then expanding and collecting factors

$$P_R(T) = \binom{L/d + N - L}{N - R}^{*}$$

$$e^{-(L+(R-L)d)T}\,(A_0 - e^{-dt}(A_1 - e^{-dt}\cdots(A_n - e^{-dt})\cdots))$$

where

$$A_n = \binom{N-R}{n} \text{ for } 0 \le n \le N - R - 1 \quad (10)$$

Equation (8) requires computer solution. In order to achieve a better intuitive grasp for the components of reliability of a modular system and to help in the initial selection of architectural organization with potentially high reliability over lengthy missions, a first estimation of the reliability of a system may be made with the aid of the graphs of Figure 4. The figure shows the reliability of modules with various degrees of redundancy against a normalized time scale. Notice the change of scale. In the first estimation of the reliability the effect of the switches can be neglected so that reliability of the system:

$$R_s = R_{CAU} \prod_{j=0}^{m} R_j \quad (11)$$

where $R_j$ is the reliability of the modules at the j–th level for all j's. The $R_j$'s may be found on the ordinate axis of the graph of Figure 4 when the normalized time has been computed.

* Use generalized factorial function i.e. $(n! = \Gamma(n+1)$

To find the reliability of a particular closed module set first compute its equivalent units of normalized time which are equal to the product of number of gates, reliability of a gate, and time; then read the corresponding reliability from the graphs. Use Equation (11) for approximate system reliability. This procedure not only allows one to work out a reasonable configuration but also indicates what the basic gate reliability must be to attain the required reliability with a feasible organization.

To illustrate the procedure by way of example, consider the configuration of Figure 2. Each level can be characterized by three parameters: N the number of modules, L those operating simultaneously and T normalized time. The system then can be summarized as:

| | CAU | AU | CU | MU | I/O |
|---|---|---|---|---|---|
| N | 1 | 3 | 3 | 3 | 3 |
| L | 1 | 1 | 1 | 1 | 1 |
| T | 2.1 | 1.49 | 3.54 | .85 | .88 |

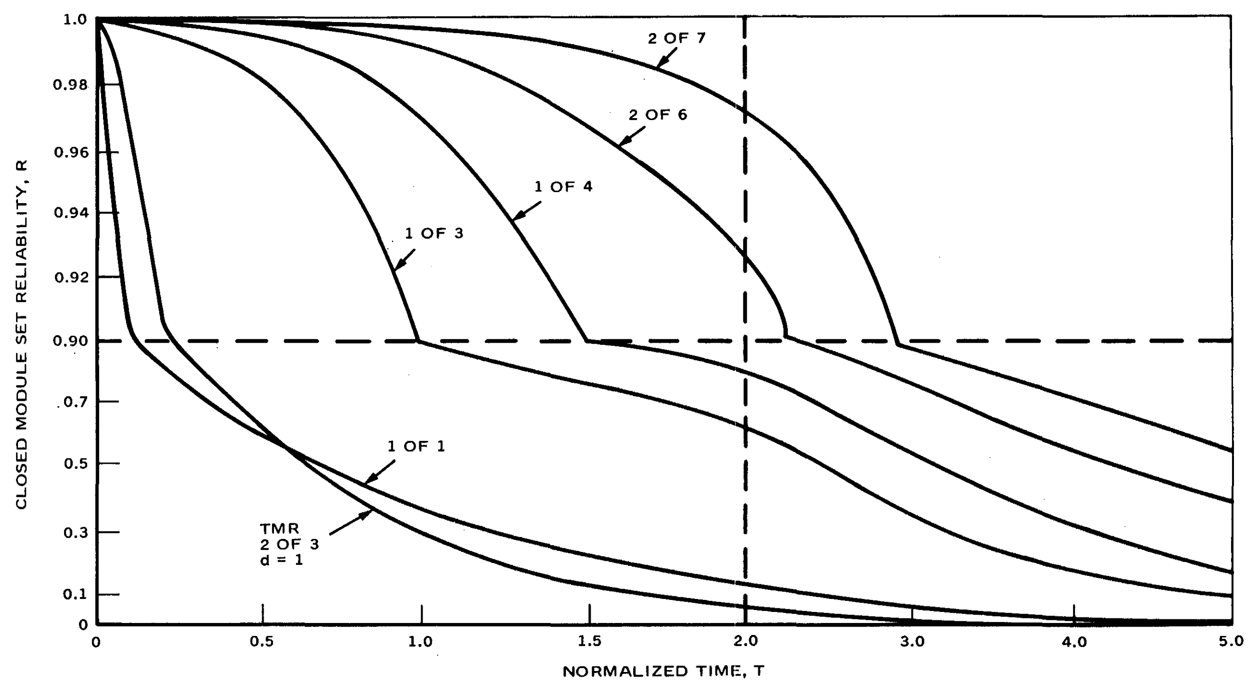
To compute normalized, gate count is obtained from Table I, the failure rate is assumed to be $\lambda = 10^{-8}$ failures per hour per gate, normalized time is computed for five years and standby failure rate is 1/10 of active failure rate (d = .1). Module set reliabilities are read from the appropriate curves of Figure 4 as follows:

| | CAU | AU | CU | MU | I/O |
|---|---|---|---|---|---|
| T | 2.1 | 1.49 | 3.54 | .85 | .88 |
| R | .12 | .76 | .24 | .92 | .91 |

Then system reliability is:

$$R_S = (.12)(.76)(.24)(.92)(.93) = .018$$

Changes must obviously be made for a reasonable reliability. In Figure 5 a step toward higher reliability is demonstrated through several types of changes. The AU and CU are each divided into interchangeable halves and the AU provided with four spare halves while the CU has five. One unit each is added to the I/O and MU units. The CAU has been triplicated and at the same time reduced to one half its former size The system summary is:



NOTES:

(1) EXCEPT AS NOTED d = 0.1

(2) READ 2 OF 6 AS L OF N TYP.

$$R = L/d \binom{L/d+N-L}{N-L} e^{-LT} \sum_{K=0}^{N-L} \binom{N-L}{K} \frac{(-1)^K}{L/d+K} e^{-KdT}$$

Figure 4—Long term reliability curves

| | CAU | AU | CU | MU | I/O |
|---|---|---|---|---|---|
| N | 3 | 6 | 7 | 4 | 4 |
| L | 1 | 2 | 2 | 1 | 1 |
| T | 1.05 | 1.49 | 3.54 | .85 | .88 |

Reading the graphs for module set reliability:

| | CAU | AU | CU | MU | I/O |
|---|---|---|---|---|---|
| T | 1.05 | 1.44 | 3.54 | .85 | .88 |
| R | .89 | .97 | .8 | .98 | .98 |

Thus, $R_S = .67$ – a significant gain in system reliability, though additional steps must yet be made to reach the desired reliability. When a potential configuration is reached one must determine the additional gates added to each module by virtue of the new configuration and then calculate a corrected system reliability. Usually one must iterate through several configurations many times to reach the desired reliability with a minimal gate count. At this point Equation (8) may be used for a more accurate reliability value.

## CONCLUSIONS

A method of estimating long term reliability of modular computers has been presented and two sample cases examined. In the second example 240 percent additional hardware was used to improve five year predicted reliability from .018 to .67. To this must be added the additional switches to accommodate the increased modules (from 13 in first example to 24 in second). To obtain a reliability of the order of .99 for a five year mission perhaps the additional hardware necessary

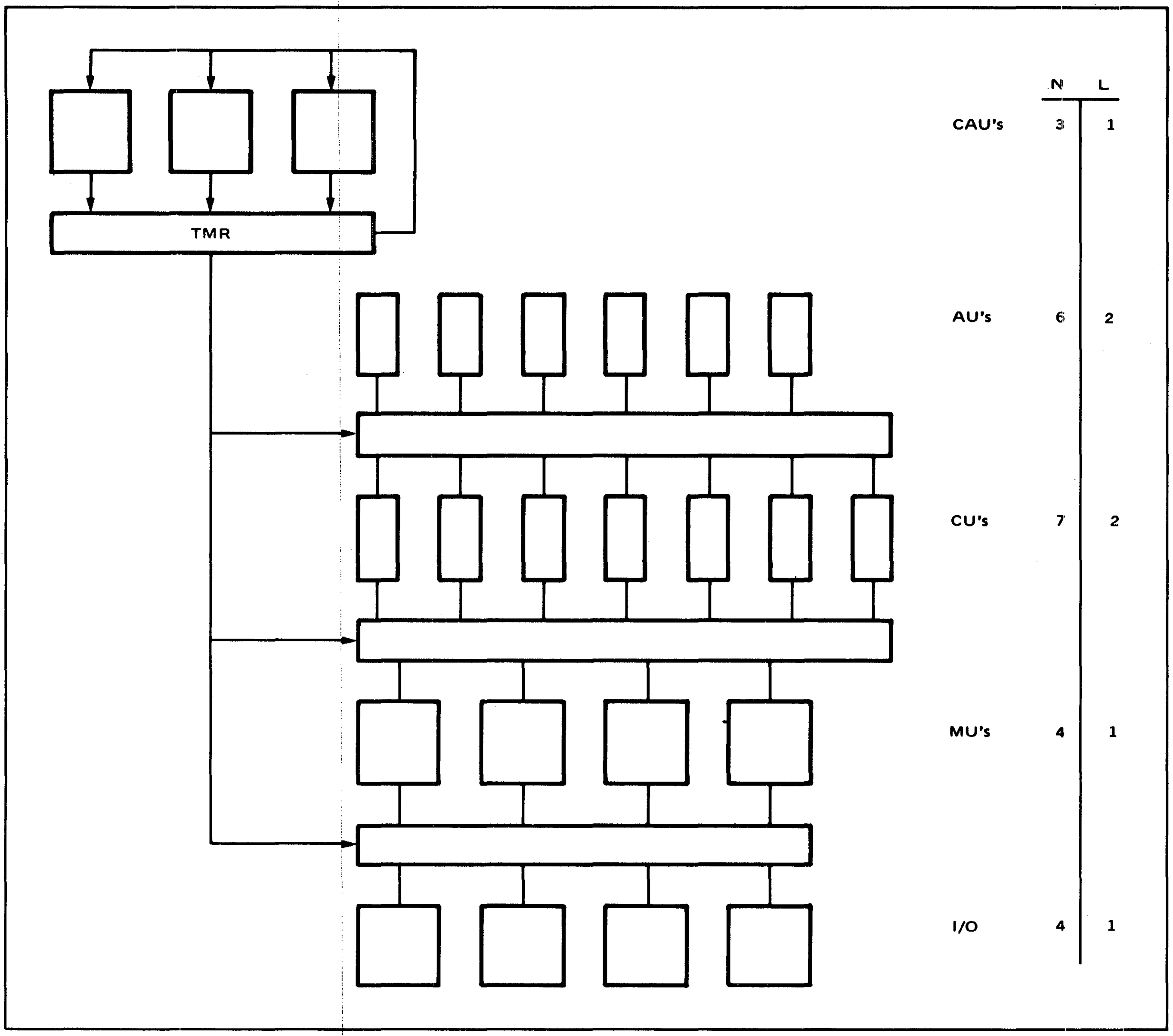


Figure 5—Multi-module modular computer

would amount to as much as four times that required for the actual computing. Gate failure rates used in the examples are for present day high quality IC's. If the basic gate reliability could be increased by a factor of ten this total additional hardware could be approximately halved.

The modular approach with standby modules appears capable of servicing long missions with feasible costs.

## ACKNOWLEDGMENT

The authors express their appreciation to Mr. Jack L. Bricker of Hughes Aircraft Company for his effort and guidance in developing the mathematical model.

## REFERENCES

1 J J PARISER H E MAURER
*Modular computer implementation with LSI*
In these proceedings
2 F D ERWIN J F Mc KEVITT
*Characters—Universal architecture for LSI*
In these proceedings
3 R A SHORT
*The attainment of reliable digital systems through the use of*

TABLE I—Two column component breakdown (approximate) modular computer breadboard (separate arithmetic & control modules)

| MODULE | GATES/MODULE | % | IC/MODULE | % |
|---|---|---|---|---|
| CAU | 4800 | 24 | 1440 | 25 |
| Switches | 180 | 1 | 55 | 1 |
| I/O | 2000 | 10 | 495 | 9 |
| Memory | 1950 | 10 | 495 | 9 |
| Control | 8100 | 40 | 2175 | 39 |
| Arithmetic | 3400 | 15 | 975 | 17 |
| TOTAL | 20430 | 100 | 5635 | 100 |
| ALTERNATE APPROACH (COMBINED ARITHMETIC & CONTROL MODULES) | | | | |
| CAU | 4800 | 25 | 1440 | 27 |
| Switches | 180 | 1 | 55 | 1 |
| I/O | 2000 | 11 | 495 | 10 |
| Memory | 2000 | 11 | 495 | 10 |
| Processor | 9800 | 52 | 2700 | 52 |
| TOTAL | 18780 | 100 | 5185 | 100 |

*redundancy—A survey*
Computer Group News March 1968

4 BERSOFF HOPE TUNG
*Modular computer research*
To be published

5 E J KLETSKY
*Upper bounds on mean life of self-repairing systems*
IRE Trans on Reliability and Quality Control Oct 1962 43-48

6 P O NERBER
*Power-off time impact on reliability estimates*
IEEE Internat Conv Record Part 10 March 22-26 1965 N Y 1-8

7 L K DAVIS G A WATSON T G SCHAIRER
*Advanced computer dormant reliability study, Final Report*
Autonetics Div of No America Rockwell Corp Oct 14 1967

8 J L BRICKER
*Reliability studies of the NASA deep space computer and the H-4400 computer*
To be published

# A compatible airborne multiprocessor

*by* E J. DIETERICH and L. C. KAYE

*RCA Aerospace Systems Division*
Burlington, Massachusetts

## INTRODUCTION

The control of large military forces is creating the need for large data-processing systems located in transport aircraft and in other situations where tight quarters and hostile environments call for the design features found in airborne systems. In these applications the configuration of the computer and its peripheral equipment strongly resembles what is found in a typical commercial data-processing system, with some additional requirements for reliability. In particular, the functional programs are complex and extensive, and the availability of a complete package of support software, including compilers and utility routines as well as the resident executive, is likely to be of critical importance. Because of its cost, so complete a software package cannot reasonably be developed specifically to answer a particular military need; it must be captured from an existing software system. The only source of complete data-management software packages is commercial data-processing; and thus it makes practical sense for a large, militarized data-processing computer to be strictly compatible with an existing commercial product. As a bonus, the commercial computer can then be used as a support computer for compilation and program checkout. An example of a program in which an airborne computer is supported by an existing ground-based commercial computer is found in the Strategic Air Command's Post Attack Command and Control System—Airborne Data Automation.[1] In this system the airborne computer is the RCA/USAF Variable Instruction Computer[2] and the ground support computer is the IBM 7090.

The hardware compatibility required for capturing system software is rigorous.[3] It is not sufficient that the militarized computer contain a large subset of the commercial instruction list, or that it obtain nearly identical results when executing the same programs. Bit for bit, the militarized computer must possess all the instructions and non-instructional features of the commercial machine, including input-output features, with the possible exception of privileged instructions usable only by the resident executive program; even here the exceptions must be few or else an entirely new executive will be required.

On long missions, especially when critical command data are being handled, the military user must have assurance that a certain minimum capability will always be available. Even with the best modern technology it is prohibitively costly to provide assured availability in a single-thread system. The classical method of coping with failure—complete duplication of the hardware, with a stand-by unit for every unit in active operation—is also unduly expensive. In most applications there are peak loads which occur relatively rarely, but which must be within the capacity of the system in its normal state, and the minimum essential capability is substantially less than the peak. What is called for is a fail-soft approach in which major components are duplicated but not allowed to remain idle. All components are used simultaneously to obtain the peak throughput, but the system can continue operation at reduced throughput in case of a failure. The failed component can be diagnosed and repaired without interrupting the operation of the surviving portions of the system and in a time short compared to the expected time to failure of the identical surviving component. Thus the user has nearly complete assurance against collapse of the entire system.[4]

*The multiprocessor hardware*

A data-processing system capable of graceful degradation is illustrated in Figure 1. Clearly, many other types of peripheral equipment could be included. All the peripheral control units are co-channelled, so that if one input-output section of the central computer should fail, another path would remain open.

The central computer, the Model 215 multiprocessor, is shown in more detail in Figure 2. It consists of two Central Processor Units (CPU), two Input–Output Units (IOU), and from two to eight Main Memory Units, interconnected by an essentially passive Signal Distribution Unit (SDU). By a conceptually simple redesign of the SDU, requiring, however, substantially more hardware, the system could be expanded to include four CPU's, four IOU's, and sixteen Main Memory Units. Each of the active units is separately powered and operates independently of other units of the same type—for instance, any number of memories can execute independent, overlapped cycles simultaneously. The SDU is merely a mechanical package housing the interconnections among the active units; as the diagram suggests, such circuits as it contains (largely line–drivers and receivers) are partitioned and powered from the active units. The logical and electrical designs conform to the constraint that a failure in any active unit, or in its partition of the SDU, must not interfere with continued operation of the remainder of the system. Multiprocessors for ground-based application

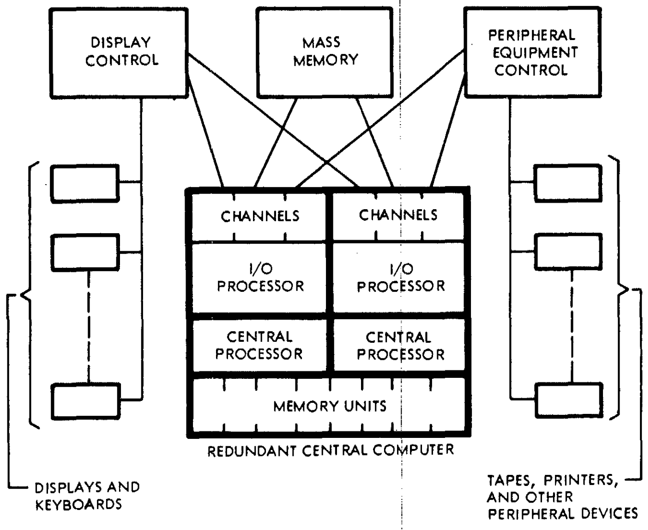


Figure 1—Typical multiprocessor application

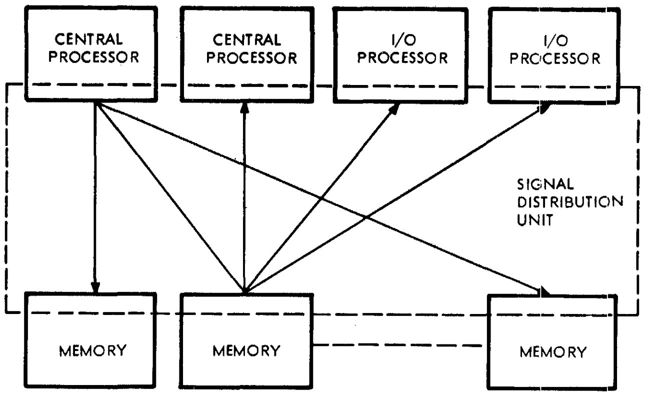


Figure 2—Fail-soft computer configuration

similar in many respects to this one have been previously described.[5,6]

If one IOU and one CPU are turned off or disconnected, the uniprocessing system that remains is functionally compatible with the RCA Spectra 70 series of commercial computers.[7,8] The entire instruction set of the Spectra 70, including privileged instructions, is contained within the Model 215 as well as the four Program States, the input-output channel control, the interrupt management scheme, and all other features of the commercial counterpart. This paper describes some of the added instructions and other features which make it possible to operate both CPU's and both IOU's together, while retaining the Spectra compatibility in the sense that any user program compiled and debugged on a Spectra 70 will run identically on the Model 215.

Either IOU can be commanded from either CPU, the choice depending only on the channel number designated in the input-output instruction, and either CPU can accept interrupts generated by any peripheral device. Except for the few microseconds when it is actually receiving a command, an IOU operates completely independently, transferring data between peripheral devices and memory without disturbing the CPU's.

Each memory unit contains 16,384 32-bit words and performs a read-write cycle in 1.65 microseconds. Input-output data rates are approximately 400,000 bytes per second for a multiplexor channel in the multiplex mode and 800,000 bytes per second for a selector channel. Each CPU executes short instructions at memory speed—for instance, an indexed add instruction in 3.30 microseconds—with single-precision multiplication in 9.0 microseconds. Except when the programs being executed in the two CPU's happen to share a memory bank, the total system throughput is twice as great as for a single CPU and IOU. The entire configuration shown in Figure 2, with eight memory units, occupies approximately 20 cubic feet, weighs approximately 1000 pounds, and consumes approximately 3550 watts. It is designed to meet the requirements of MIL-E-5400, Class I, the basic specification for airborne electronics.

The integrated circuits and medium-scale arrays used in the Model 215 are of military quality, with burn-in and screening. When the standard degradation factor for airborne application (five to one) is applied, the calculated mean time to failure of a unit (including an associated partition of the SDU) is 2200 hours for a Memory Unit, 1305 hours for a CPU, and 1820 hours for an IOU with a maximum channel capacity. The only single-thread element in the entire computer is the master oscillator, which is located in the SDU and redundantly powered by both CPU power supplies and which has an airborne MTBF of 100,000 hours. Use of redundant oscillators with a voting circuit was considered, but the calculated failure rate of the single-thread portion of the voting circuit turned out to be higher than that of a single oscillator.

The purpose of fail-soft features is to permit an airborne mission to be completed successfully, even though portions of the hardware might fail. For a 72-hour mission which can be successfully completed with the minimum capability provided by one CPU, one IOU, and six out of eight Main Memory Units, the probability of successful completion is greater than 99 percent, even if no repair is possible during the mission.

Recently there has been a great deal of activity aimed at producing concepts for software, and for the supporting hardware, which will facilitate parallel processing of tasks and portions of tasks.[9,10,11] In one respect, the present work is much less ambitious, as no special hardware has been incorporated for the purpose of forking and joining parallel processes within a user program. In other respects, however, it is more ambitious, in that a rigorous attempt has been made to avoid single-thread hardware of any kind, and in the commitment to achieve the goals of multiprocessing and graceful degradation while capturing a complete software package designed for a family of computers not having these features. Certain aspects of the executive software require special attention in order to realize the benefits of multi-processing, and the hardware must be designed from the outset with these requirements in mind. In normal operation the most significant topics are the control of the executive, the management of input-output interrupts, the assignment of CPU's to working programs, and initial loading. Problem recovery and self-diagnosis are the critical aspects of graceful degradation.

### *Executive control*

In order to preserve the purity of the fail-soft features, the Model 215 hardware is completely symmetrical, with both CPU's and both IOU's identical. It is therefore desirable for the executive program to avoid creating a master-slave relation between the CPU's. In fact, it is convenient to regard both CPU's as slaves, with the resident executive program as the master. Because programs checked out on the commercial support computer must run correctly on the Model 215, the executive must present the user program with precisely the same interface as the executive in the support computer and must allocate resources and manage input-output in a unified way for all programs. Because the executive supports the user programs in many complex ways, the multi-processing executive should be derived with minimum possible modification from an existing, commercial executive, so as to introduce the least possible chance of mismatches at the interface. The Tape-Disc Operating System[12] of the Spectra 70 is an excellent example of a software system which has desirable characteristics for military data-management and which contains a multi-programming executive readily adaptable for multi-processing.

The executive program must be protected from having its coding executed by two CPU's at once. Even if the coding were parallel reentrant, there are common tables carrying the status of programs and of input-output devices which can become garbled if two CPU's are allowed access at the same time. The executive can be partitioned into independent subroutines, so that different portions may be executed simultaneously; nevertheless, the system must provide means for one CPU to lock the other one out of the subroutine it is currently executing.

An ideal mechanism for locking out a CPU is the Test and Set instruction.[13,14] This instruction tests a specified byte in core memory and simultaneously sets it to all ones. If two CPU's attempt to execute this instruction on the same byte at the same time, the one having higher priority for access to memory will perform the operation first; if the tested byte was initially not set, the higher priority CPU will detect that fact, but the other CPU will not have access to the test byte until after the byte has been altered.

Figure 3 is a flow chart showing how the Test and Set instruction is used for executive lockout. The initial instruction of each independent subroutine is a Test and Set, followed by a Branch to direct the CPU in accordance with the previous setting of the test byte. There is a unique test byte for each independent subroutine. If the test byte was not originally set, the coding is available for execution and the program proceeds normally. Another CPU arriving at the Test and Set instruction immediately thereafter will find the test byte set and will branch to the Timed Idle instruction. The first CPU meanwhile completes the disputed section of coding, restores the test byte to its original form, and then executes an instruction to terminate the idle condition of the other CPU. This privileged instruction, a special feature of the Model 215, causes a signal to be sent out to terminate any Timed Idle instruction then being executed. The idling CPU then returns to the Test and Set instruction and proceeds into the previously locked-out subroutine.

If the active CPU fails in such a way as to go into a loop, the idling CPU will never receive a signal to terminate the idle mode. The time-out feature is provided in order to prevent "silent death" in this case. The duration of the idle mode can be adjusted in accordance with the expected time to execute the subroutine being protected. The idle mode terminates with a unique condition code if it times out instead of being terminated by a signal from the other CPU; the executive can then record the suspicion of a malfunction in the other CPU before continuing with normal processing or going to an error recovery routine.

*Input-output interrupts*

In the Model 215 either CPU must be able to command peripheral devices through either Input-Output Unit. A user program may be executed partly by one CPU and partly by the other; yet it must be able to have access to peripheral devices on any channels, whichever Input-Output Unit may contain those channels. Certainly in case of CPU failure the surviving CPU must have access to all channels. When an input-output operation is completed, the peripheral device generates an interrupt signal calling for CPU action. In general the program which called for the input-output operation may have been forced to wait for its completion, and at the time of the termination interrupt no CPU will be executing it; thus, there is little reason to tie the servicing of a termination interrupt to the CPU which initiated the input-output action. Again, the requirement for continued system operation in the face of failures demands that either CPU be able to respond to an interrupt coming from either Input-Output Unit.[15]

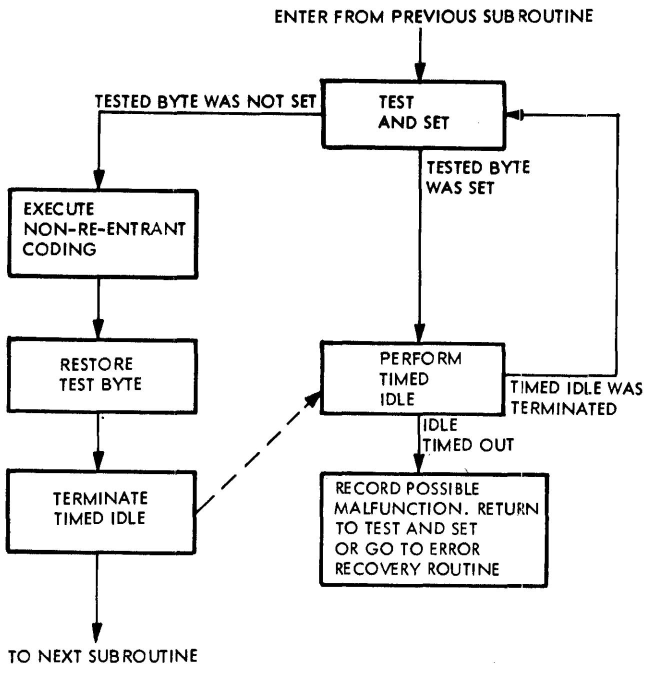

ENTER FROM PREVIOUS SUBROUTINE
TESTED BYTE WAS NOT SET
TEST AND SET
TESTED BYTE WAS SET
EXECUTE NON-RE-ENTRANT CODING
RESTORE TEST BYTE
PERFORM TIMED IDLE
TIMED IDLE WAS TERMINATED
IDLE TIMED OUT
TERMINATE TIMED IDLE
RECORD POSSIBLE MALFUNCTION. RETURN TO TEST AND SET OR GO TO ERROR RECOVERY ROUTINE
TO NEXT SUBROUTINE

Figure 3—Executive lockout

The processing of an input-output interrupt requires action from both a CPU and an Input-Output Unit. The CPU must execute a portion of the executive which maintains tables carrying the status of peripheral devices and which updates the readiness status of the affected program. In order to do this, the executive needs information on the status of the peripheral device in question and of the channel through which it worked. At the time the interrupt is taken this information is obtained by the Input-Output Unit on command from the CPU and is stored in addressable registers in the Input-Output Unit. The executive gains access to the information by a Store Scratchpad instruction addressing the registers of the appropriate channel. Significant deviations from this pattern are prohibited by the requirements of compatibility.

Although either CPU must have the ability to respond to any input-output interrupt, clearly both CPU's must not respond to the same interrupt. For one thing, duplicate updating of the executive tables would be improper; although this could be prevented by judicious use of the Test and Set instruction, a great deal of time would be wasted in unnecessary housekeeping. Moreover, in the very act of accepting the interrupt the CPU automatically commands the Input-Output Unit to obtain and store the device status information, and this cannot be done twice for the same interrupt without losing data. Flexibility of response must be achieved without duplication of activity.

In the Spectra 70, with which the Model 215 is compatible, interrupt signals reported to the CPU are stored in an Interrupt Flag Register. The Interrupt Flag Register contains a bit for each input-output channel, plus some additional bits for interrupts not related to input-output. There is also an Interrupt Mask Register, controllable by the executive program by means of privileged instructions; the contents of the Interrupt Mask Register designate specific interrupts to be taken as soon as they are requested and others to be held until it is convenient for the executive program to respond. Normally, all input-output interrupts are taken promptly unless the executive is already responding to an interrupt.

The Model 215 requires a software convention to establish which interrupts will be taken by each CPU. This convention may be arbitrary, so long as each input-output interrupt is taken by one and only one CPU. An example of such a convention is illustrated in Figure 4. The executive program sets the Interrupt Mask Register of each CPU to take the interrupts designated for that CPU, and the hardware reports all input-output interrupt signals to both CPU's.

When an interrupt is taken by the designated CPU, the corresponding bit in the Interrupt Flag Registers of both CPU's is reset. At all times both Interrupt Flag Registers contain a record of all the pending interrupts. In case of failure of one CPU, the executive program, as part of the recovery process, can alter the Interrupt Mask Register of the surviving CPU so as to permit that CPU to take all input-output interrupts.

In the simplest case the Interrupt Mask Registers of both CPU's would be given permanent settings when the system is initially loaded, and these settings would only be altered in case of a CPU failure. If desired, however, the executive program can adjust the Interrupt Masks of the two CPU's in accordance with the priorities of the programs being executed, so that high priority programs are not interrupted.

After an interrupt has been taken and the Input-Output Unit has stored the device status information in the appropriate channel registers, the contents of these registers must be protected until the CPU taking the interrupt has performed enough analysis to identify the registers to be stored and to execute the instruction to store them. In the meantime, the other CPU, executing a different subroutine in the executive, may attempt to start a peripheral device on the same

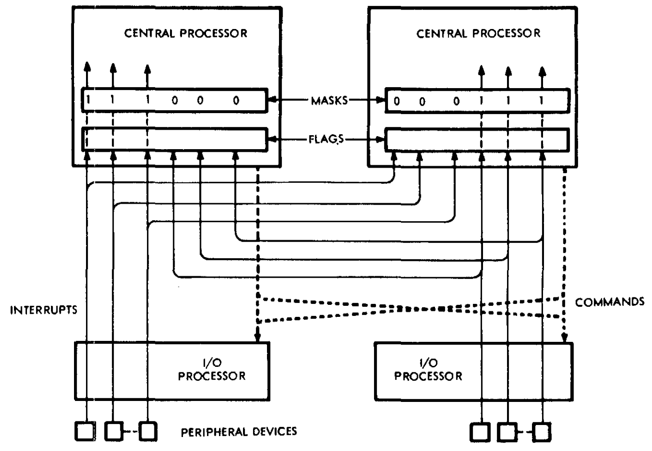


Figure 4—Input-output control block diagram

channel (or subchannel in the case of multiplexed devices). This could destroy the information related to the interrupt. An interlock is therefore incorporated, preventing the other CPU from executing a Start Device instruction until the channel registers have been stored. The interlock carries a time-out feature, in case the CPU taking the interrupt goes into a loop before storing the channel registers; if the interlock times out, the interlocked CPU is informed by a special interrupt that a malfunction may have occurred. A similar situation arises when an attempt to start a peripheral device fails because of a malfunction or other peculiar behavior of the device; status information is again stored in the channel registers and must be protected from destruction by the other CPU, and the same interlock is invoked.

## *Load balancing*

When the Model 215 is operating normally, two or more object programs, as well as the executive, are resident in core, and each CPU is executing one of them, independently of the other. From time to time one of the CPU's may be diverted by an interrupt from the processing of an object program; it will then spend some time executing a portion of the executive. When the interrupt processing is completed, the interrupted CPU should return to processing an object program. The executive must insure that each CPU has productive work to do, and that the highest priority object programs are being processed.

An analogous situation arises in a computer without a multi-processing feature if the executive is capable of multi-programming. Here again there are several resident object programs in addition to the executive.

At a given time, some of these programs may be forced to wait for the completion of input-output, for loading of additional program segments, or for other special action on the part of the executive. It is the responsibility of the executive to ensure that the highest priority program capable of running is entered whenever the executive itself does not require the use of the CPU.

The normal flow of a multi-programming executive is suggested by the flow chart of Figure 5. In the steady state the CPU executes an object program until an interrupt is taken. The interruption may be caused by the object program itself—typically a Supervisor Call to start an input-output operation—or by an external agency, as in the case of a termination interrupt from a peripheral device. In the first case, the program interrupted may be unable to continue until some action initiated by the executive has been completed; in the second case, the processing of the interruption may change some other program from a waiting status to the status of being ready to run.

The multi-programming executive maintains an Operation List, a table of vital information about each program resident in core. In the Operation List for each such program there is a set of flags indicating the reason why a particular program is not ready to run. The flag bits are updated, as appropriate, by the processing of interrupts which affect the readiness of programs. If no flag is set, the program in question is ready, and the executive, when it has completed its other processing, exits by giving control of the CPU/ to the highest priority program that is ready. At that time the executive stores the identity of this program in the table of Current Operations, to establish an information trail in preparation for subsequent interrupts.

This executive exit structure can be converted into a form suitable for multi-processing control by two simple changes. An additional flag bit is needed in the Operation List for each program; this bit tells the executive exit that the program is not only ready but is already being executed, and is updated each time a program is activated or interrupted. In the Spectra executive an unused bit is available for the purpose in the format of the flag byte in the Operation List. The other change is to carry an entry in the table of Current Operations for each CPU in the system. Whenever a CPU passes through the executive exit, the appropriate entry is updated. The only special hardware needed is a means for the otherwise identical CPU's to identify themselves to the executive, so that the proper entry in the table of Current Operations may be used. This means is provided by an instruction

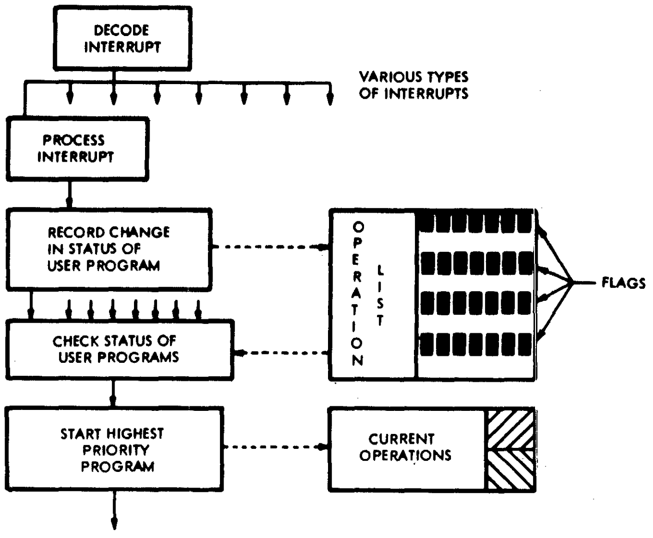

DECODE INTERRUPT
VARIOUS TYPES OF INTERRUPTS
PROCESS INTERRUPT
RECORD CHANGE IN STATUS OF USER PROGRAM
CHECK STATUS OF USER PROGRAMS
START HIGHEST PRIORITY PROGRAM
OPERATION LIST
FLAGS
CURRENT OPERATIONS

Figure 5—Load balancing

which stores the CPU identity, derived from its plug-in position at the SDU, in a specified General Register.

The multi-processing modification to the executive exit can be extended to handle as many CPU's as desired merely by adding to the size of the table of Current Operations; additional flag bits in the Operation List are not needed. The instruction permitting a CPU to identify itself must, of course, store enough bits to identify the number of CPU's in the system.

A different sort of refinement permits the executive always to maintain all the CPU's at work on the highest priority programs that are ready, even if the CPU taking an interrupt, and therefore passing through the executive exit, was itself executing a high-priority program when interrupted. Suppose, for instance, that three programs are resident. The highest priority program is waiting for the completion of an input-output operation, and when the termination occurs, the CPU executing the higher priority of the two remaining programs takes the interrupt. In the normal course of events, the highest priority program would be designated as ready, and the CPU which took the interrupt would begin to execute it as soon as interrupt processing was completed. The result would be that the highest and lowest priority programs would be running, while the one with intermediate priority would be ready but would have to wait for a CPU to become available.

For reasons connected with the requirement that system processing continue in the face of failure of a CPU, so long as one CPU survives, it is necessary for the system to incorporate an instruction permitting one CPU to gain the attention of the other. This privileged instruction causes an interrupt in the CPU being signaled. Using this signal, the executive can examine the priorities of all the programs currently being executed, and can interrupt a CPU which is executing a program lower in priority than some program not being executed at the time. A situation calling for such examination can only arise as a result of processing an interrupt; so provision for this extended priority surveillance would be a natural extension to the multi-processing modifications to the executive exit.

*Initial program loading*

Initial loading for a multi-processor differs little from initial loading of a conventional computer having a multi-programming executive. The operator's console has provision for selecting the device and channel through which the bootstrap routine will be loaded. A selected CPU is then started at a fixed address, where it encounters coding that initializes its General Registers, Interrupt Mask Register, etc. Next the resident executive is loaded, followed by a number of object programs appropriate to the amount of core memory available. For multi-processing to be effective, there must be at least two resident object programs; and to avoid inefficiency caused by waiting for input-output terminations, preferably three or four. When the memory is loaded, the selected CPU initializes the program counter of the other CPU and commands it to start. One after another, the CPU's execute the normal coding of a multi-programming executive to commence execution of object programs. The extra flag bits in the operation list ensure that only one CPU will execute a given object program.

The hardware requirements attributable to the multiprocessor configuration are the provision for selecting one of the CPU's to execute the bootstrap coding, means for enabling one CPU to set the program counter of the other and command it to start, and means for a CPU to identify itself to the executive so that the proper Interrupt Masks may be established.

*Problem recovery*

In order for any system to continue in operation after a failure in one of its units, there must be sufficient checking built into the hardware. A failure must be detected before it has had a chance to propagate errors far into the problem being solved at the time of failure. The Model 215 Main Memory Units are checked by the usual byte parity on the data, and by a parity bit in the key memory which forms part of the memory protection feature. The Input-Output Units employ parity checking on data and on command words. In the CPU's the scratch-pad memory and the elementary operations stored in the read-only memory each contain parity bits. The arithmetic unit is two words wide, to speed up byte-oriented and double-precision instructions and to simplify their control; when operating on single-precision data, the two halves of the arithmetic unit work in parallel and the results are compared. Whenever possible, data transmitted between units has a parity check at both ends of the transmission path to facilitate diagnosis of faults. A small number of special checking circuits are incorporated to check for faults not detectable by parity checking or arithmetic comparison.

Another requirement for continuing operation is the preservation of enough information to permit resumption of the program by the surviving units. The necessary information, if it has not been destroyed by the failure itself, will be found partly in memory and partly in various processing registers. In general, this information cannot be made available to an operator without the execution of some operations by the surviving processors; in other words, purely manual recovery is impossible. Since the hardware and software must include the capability of retrieving and identifying the information needed to continue processing, it is only a short step to providing for completely automatic problem recovery in the majority of cases.

Failures in the Input-Output Units and the peripheral equipment present no problems unique to multi-processing. Data-transfer operations may be retried under program control. If the failure persists, or if the error information stored in the Channel Status byte indicates that the failure is in the Input-Output Unit rather than in the peripheral equipment, the peripheral device may be switched to a channel on the other Input-Output Unit; in the case of co-channelled devices the switching consists merely of making an entry in an executive table.

When a failure occurs in a CPU, either of two modes of operation illustrated in Figures 6 and 7 may be followed. Normally there is a surviving CPU which can come to the rescue. In this mode the failed CPU stops dead as soon as the failure is detected, thus preserving the contents of its processing registers for problem recovery and fault diagnosis. In the act of stopping it sends a signal to the surviving CPU; this signal causes an interrupt into Program State P-4, the

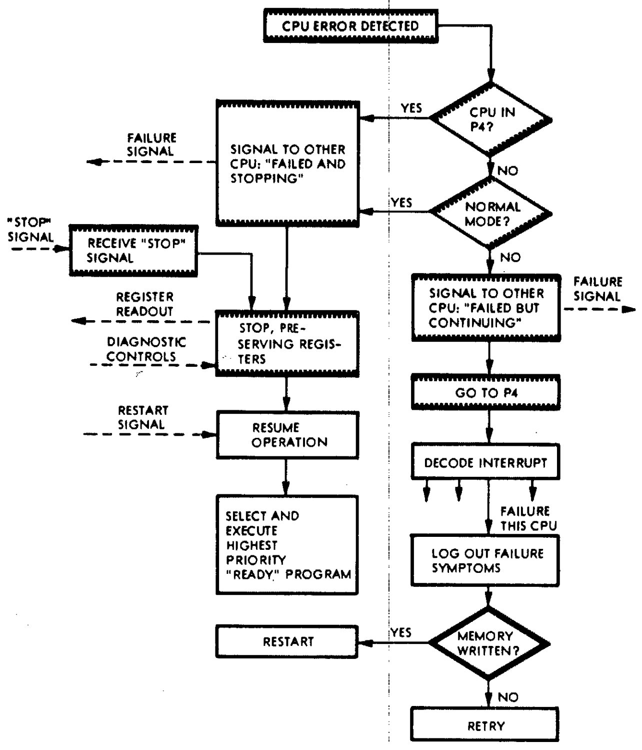


Figure 6—Central processor error: Failed CPU

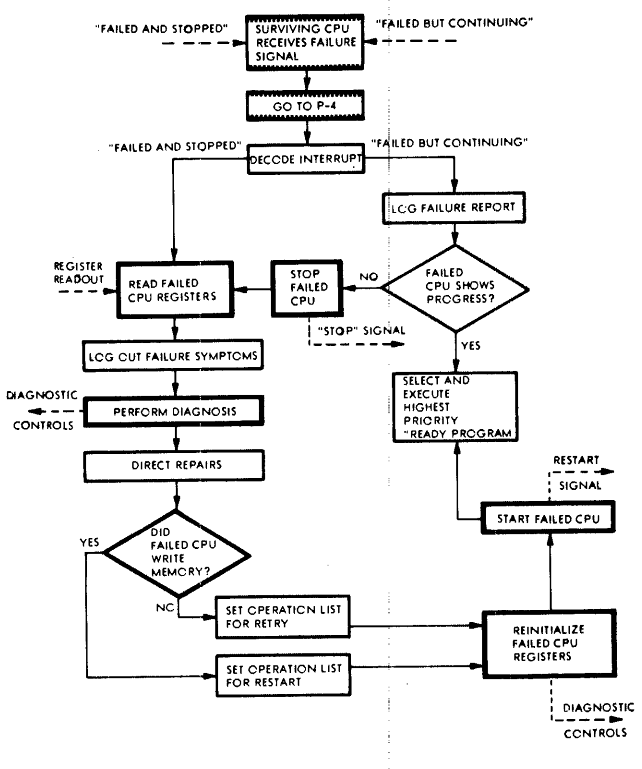


Figure 7— Central processor error: Surviving CPU

normal state for responding to machine failure. It also provides an indication, accessible to the program, that the reason for the interrupt was a CPU failure not in the CPU taking the interrupt. After decoding the reason for the interrupt, the surviving CPU can read out selected processing registers of the failed CPU and determine its status. One bit available in this way indicates whether or not the failed CPU had written into its scratchpad or into Main Memory before stopping. If not, all the data needed to repeat the instruction in progress at the time of failure is still available, and the program can be continued from that point. If data have been destroyed, the program must be returned to a restart point, exactly as in a conventional computer, but with the advantage that the restart can be initiated without human intervention if desired.

A second mode of operation is available to deal with a failure in the sole surviving CPU. Such failures will be rare, since the mean time to repair is very small compared to the mean time between failures; nevertheless, because a large proportion of errors in any computer are normally transient, it is well to allow the CPU to attempt its own recovery. In this mode, which the executive can establish when its configuration table reveals that a particular CPU is the sole survivor, the failed CPU takes an interrupt to Program State P-4 and continues processing. If the error was transient, the CPU will be able to decode the interrupt and will then determine whether the program can be resumed or must be restarted. If, on the other hand, the failure is solid, the CPU will commit another error. This time, since the error indication is a failure of the CPU while in Program State P-4, the CPU stops dead, just as it does in the normal mode. Naturally, the problem recovery portion of the executive must not contain double-precision instructions whose arithmetic is not checked.

The flow charts of Figures 6 and 7 present condensed pictures of this process. The shaded boxes represent actions taken automatically by the hardware; the other boxes represent executive program action. Special instructions are provided for the purposes indicated in the unshaded boxes with heavy borders. The dotted lines represent direct wire connections between the two CPU's. In the normal mode, the surviving CPU takes control of the entire process as soon as an error is detected. It may enter a diagnostic routine after logging out the registers of the failed CPU; alternatively, the problem recovery routine may simply record the failure, reinitialize the registers of the failed CPU and turn it back on, in the hope that the error was transient. In either case, the surviving CPU determines whether the program can be resumed or must be restarted, sets the program entry point accordingly, and updates the Operation List flags for that program, so that it can be taken up by some CPU in accordance with its priority.

In the other mode, of course, the failed CPU attempts to do all this for itself. The CPU that did not fail can observe the progress of the failed CPU by monitoring appropriate locations in the executive portion of memory; if these locations are not being properly updated, the failed CPU can be stopped and the processing used in the normal mode can be undertaken.

Failure in a main memory unit is detected as a parity error and is reported to the CPU or Input-Output Unit currently working with the failed memory unit. An Input-Output Unit stores the error indication in the Channel Status Byte and treats it exactly as a Spectra 70 does. If the parity error occurred during execution of an instruction, the associated CPU treats the error as illustrated in Figure 8. The other CPU is not notified, since the CPU receiving the error indication is presumably operable.

As in the Spectra 70, the CPU takes an interrupt to P-4. After decoding data stored automatically in taking the interrupt, and determining that the cause was a memory parity error, the executive locates the instruction which caused the error. The Program Counter and Instruction Length Counter provide the necessary information. By means of a special instruction—Check Parity—the parity error is localized to the instruction word or an operand. The Check Parity instruction moves data by words, correcting the parity of any error byte it encounters. The instruction terminates automatically if a parity error is encountered and identifies the location of the error byte. It also indicates by a condition code whether or not it was completed without a parity error, but it does not cause an interrupt if a parity error occurs.

Having located the parity error, the executive can determine whether its own memory bank is affected. If the error is not in the executive bank, then problem recovery concerns only a single user program. This program can be returned to a restart point in much the same way as in a conventional computer. The affected memory bank may be tested by a diagnostic program, and if the error does not appear to be transient, the bank may be removed from the system. Compatibility with existing software requires that the available memory be organized with consecutive ad-

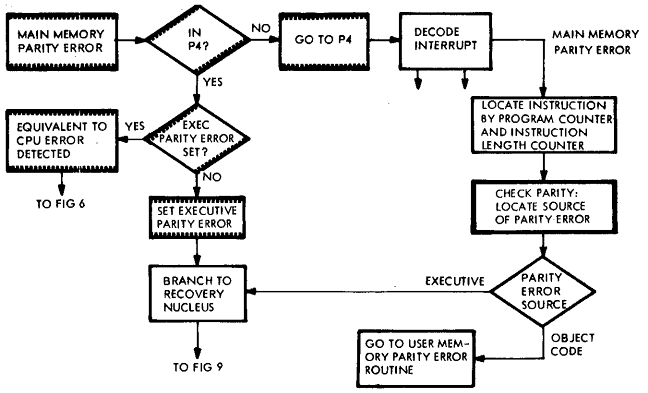

MAIN MEMORY PARITY ERROR
IN P4?
NO
GO TO P4
DECODE INTERRUPT
MAIN MEMORY PARITY ERROR
YES
LOCATE INSTRUCTION BY PROGRAM COUNTER AND INSTRUCTION LENGTH COUNTER
EQUIVALENT TO CPU ERROR DETECTED
YES
EXEC PARITY ERROR SET?
NO
TO FIG 6
SET EXECUTIVE PARITY ERROR
CHECK PARITY: LOCATE SOURCE OF PARITY ERROR
BRANCH TO RECOVERY NUCLEUS
EXECUTIVE
PARITY ERROR SOURCE
OBJECT CODE
GO TO USER MEMORY PARITY ERROR ROUTINE
TO FIG 9

Figure 8—Executive parity error

dresses, although obviously with a failed bank the total amount of memory is reduced. The Model 215 has a special instruction which permits the executive to reassign memory bank addresses. In this way, the system can continue to process as many programs as it can fit into surviving memory, regardless of which bank fails.

If, on the other hand, failure occurs in the memory bank containing the executive, special features are required in order to prevent the collapse of the entire system. The goal for the Model 215 is to confine the problem to the one program being serviced by the executive at the time of the failure. When the executive bank has failed, it may be impossible to execute even a single instruction after the interrupt is taken, and so the interrupt cannot be decoded. If this occurs in the Model 215, the hardware will automatically take a branch to the contents of a fixed General Register of Program State P-4. This branch will normally point to a Recovery Nucleus located in a different memory bank. If still another parity error should occur, it would be an indication that the failure was really in the CPU, and appropriate action would be taken.

The Recovery Nucleus, Figure 9, consists of copies of critical, dynamic executive tables, plus enough coding to load a fresh copy of the executive into a surviving memory bank. The Recovery Nucleus contains configuration tables showing which units are still operable. After identifying the memory bank into which it will load the executive, the Recovery Nucleus must quiet all peripheral devices in order to prevent accidental overlaying of the new executive as it is loaded. A special instruction is provided so that peripheral devices may be quieted without destroying data or taking the termination interrupts.

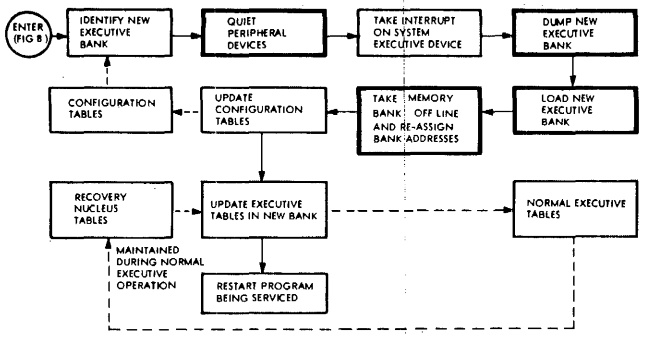

ENTER (FIG 8)
IDENTIFY NEW EXECUTIVE BANK
QUIET PERIPHERAL DEVICES
TAKE INTERRUPT ON SYSTEM EXECUTIVE DEVICE
DUMP NEW EXECUTIVE BANK
LOAD NEW EXECUTIVE BANK
TAKE MEMORY BANK OFF LINE AND RE-ASSIGN BANK ADDRESSES
UPDATE CONFIGURATION TABLES
CONFIGURATION TABLES
RECOVERY NUCLEUS TABLES
UPDATE EXECUTIVE TABLES IN NEW BANK
NORMAL EXECUTIVE TABLES
MAINTAINED DURING NORMAL EXECUTIVE OPERATION
RESTART PROGRAM BEING SERVICED

Figure 9—Recovery nucleus

If there is an interrupt on the device which carries the backup executive, that one interrupt must be taken. The current contents of the memory bank which is to accept the new executive are dumped, in order to avoid losing any programs which use it, and the new executive is loaded. A special feature of the Start Device instruction allows the Channel Address Word to be taken from any bank. The failed bank is taken off line, and memory banks may be readdressed if required. After updating the configuration tables and transferring information from the Recovery Nucleus tables to the new executive tables, normal operation can be resumed.

### *Self-diagnosis*

On missions of long duration, the ability to locate and repair faults during the mission is necessary in order to give high assurance that the minimum essential capability will always be available.

The fail-soft features of the Model 215 rest on the assumption that two failures do not occur simultaneously—that is, that a second failure does not occur before the first failure has been repaired. The mean time to repair of the Model 215 will be less than thirty minutes. To make this possible, without requiring high skill levels, the Model 215 is packaged and powered so that a unit may be repaired while the system is operating, the logic is partitioned into functional, replaceable cards, and there is provision for automatic fault diagnosis.

Ease of access and functional partitioning are matters of straightforward engineering design. Likewise, the software techniques for automatic diagnosis of memory failures are well-known and require no special hardware. Fault isolation in an Input-Output Unit does require some special hardware, chiefly to isolate peripheral equipment from the faulty unit so that it can be used with the surviving Input-Output Unit, and a means of simulating signals from the peripheral devices in order to exercise the unit to be diagnosed.

The real challenge is CPU failure. Even here, with well thought out functional partitioning, isolation of a fault to one or two functional cards is not too difficult if the processor can run a program. Unfortunately, the "hard core" of equipment that must be operable in order for a diagnostic program to run can amount to as much as 70 percent of the CPU. Diagnosing the hard core requires some form of external stimulus-and-measurement equipment. In the Model 215 the stimulus-and-measurement equipment for a failed CPU is the other CPU.

In the normal mode of operation, as illustrated in Figure 6, detection of an error (except for a Main Memory Parity Error) causes the offending CPU to stop immediately, alerting the other CPU to the trouble. The failed CPU stops as soon as the error is detected, so that the contents of its scratchpad memory and its processing registers are undisturbed. By means of a small number of direct-wire connections between CPU's, the surviving CPU can gain access to the preserved information. It then adopts the so-called "start small" technique of diagnosis.[16] The failed CPU can be commanded to put the contents of its memory buffer register onto the main inter-unit bus for examination by the survivor and for storage in core. Data can also be written into the memory buffer. By cycling a few bit patterns through the memory buffer the diagnostic program can localize a fault within the memory buffer or can determine that the memory buffer is operable. In the first case the problem is solved; in the second case the memory buffer can be used as a dependable tool to explore further into the failed processor's hard core.

Other direct connections allow the contents of the read-only control memory to be read out to core for checking and also permit the surviving processor to command the failed processor to execute specific elementary operations. In this way the contents of the processing registers can be examined and checked. In some cases the failure can be identified at once. For instance, if the error was in the arithmetic of a single precision operation, it will have been caught by the comparison check between the two halves of the arithmetic unit. The operation can be simulated step-by-step on the surviving CPU, while the failed CPU is driven through the operation by individually activated elementary operations. When the simulation differs from the actual operation in one or the other half of the failed arithmetic unit, the simulation program can identify the bit position in which the fault originated. Since the arithmetic unit is partitioned into four-bit slices, with all the arithmetic registers on the same card, the proper card to replace is obvious at once, and the operator can be notified.

If all the registers and transfer paths of the failed CPU are verified by this step-by-step exercise without locating the trouble, the hard core can be assumed to be working, and the failed CPU can be commanded to execute its own diagnostic programs. This will frequently occur when the failure is intermittent or data-sensitive. Considerable field experience will be needed in order to determine the optimum point for turning over the fault-isolation process to the failed processor. Putting the failed processor back on its feet, either for executing its own diagnostic routines or following repair, requires setting the Program Counter and commanding the processor to start executing instructions. The presence of the first capability is implied by the diagnostic process of loading and reading processing registers; the second is already needed for initial program loading.

Whether the failed processor or the survivor is doing the work, the diagnostic programs can be entered into the Operation List and executed as if they were user programs. In this way, the diagnostic process can be carried along on a time-sharing basis in parallel with mission data processing. By adjusting the priority of the diagnostic program, the operator or the executive program can react to the relative urgency of accomplishing specific mission tasks as compared with restoring full processing capacity.

## SUMMARY

A multi-processor with independent Central Processors, Input-Output Units, and Memories can provide graceful degradation as well as sufficient compatibility with a commercial computer to capture its entire operating system. Computing load and the management of input-output operations are balanced without establishing a master-slave relationship between Central Processors. Either Central Processor can diagnose the other one. By a combination of special hardware and software features the system is able to continue operation in the presence of failures, including failures in the executive bank of memory.

## REFERENCES

1 E H MILLER E J DIETERICH P T FRAWLEY
*The post attack command and control system—Airborne data automation (PACCS-ADA), 481A*
Proc Natl Aerospace Electronics Conf 1969

2 E H MILLER
*Reliability aspects of the RCA/USAF variable instruction computer*
IEEE Trans on Electronic Computers Vol 16 No 5 1967 596-602

3 J A WARD
*A panel discussion—Software transferability*
Proc SJCC Vol 34 1969 605-612

4 R P HASSETT E H MILLER
*Multithreading design of a reliable aerospace computer*
IEEE Trans Aerospace Electronics Systems Vol 2 1966 147-158

5 R A MERIKALLIO F C HOLLAND
*Simulation design of a multiprocessing system*
Proc FJCC Vol 33 Part 2 1968 1399-1410

6 J F KEELEY
*An application-oriented multiprocessing system*
IBM Systems Journal Vol 6 No 2 1967 78-79

7 A D BEARD
*RCA Spectra 70 basic design and philosophy of operation*
Proc WESCON 1965

8 A T LING
*The scratchpad oriented design of the RCA Spectra 70*
Proc FJCC 1965

9 M E CONWAY
*A multiprocessor system design*
Proc FJCC Vol 24 1963 139-146

10 J B DENNIS E C VAN HORN
*Programming semantics for multiprogrammed computations*
Comm ACM Vol 9 1966 143-154

11 M LEHMAN
*A survey of problems and preliminary results concerning parallel processors*
Proc IEEE Vol 54 1966 1889 1900

12 RCA Information systems, Spectra 70 Tape-Disc Operating System, Control System Reference Manual 1967 70-00-611

13 IBM-System/360 Principles of Operation S 360-01
Form A22-6821-2 1966 7 74-75

14 RCA Spectra 70/46 Reference Manual 70-46-601 1967 171

15 R J GOUTANIS N L VISS
*A method of processor selection for interrupt handling in a multiprocessor system*
Proc IEEE Vol 54 No 12 1996 1812-1819

16 J J DENT
*Diagnostic engineering requirements*
Proc SJCC Vol 32 1968 503-507

# Large-scale integration: Promises versus accomplishments—The dilemma of our industry

*by* H. G. RUDENBERG

*Arthur D. Little, Inc.*
Cambridge, Massachusetts

SUMMARY

This paper discusses the dilemma posed by the promises made about large-scale integration, and the expectations derived from the promises. Furthermore, it examines LSI's present form. In some instances what have appeared to be "broken promises" are not in fact that at all. Some believers wanted to believe and thus have suffered from self-delusion. Some promises certainly were unwise or premature, thus creating false impressions. But others represented a misunderstanding between component and system engineers.

The paper both analyzes and interprets the history of the technical developments that led from transistors to integrated circuits to large-scale integration, describes some of the pressures that have led to premature promises, and characterizes the technology leading to LSI. Likewise, it looks at the introduction dates of systems, as well as of the components that have implemented such systems in the past. This review covers such integrated circuits as gates or amplifiers to LSI-type digital differential analyzers, which can perform a variety of calculations and logic, or, in another example drawn from the memory field, from shift registers to scratch pad and larger memories using semiconductor arrays. Likely near-term uses of LSI devices in new computer systems include the high-speed buffer memory in the 360 Model 85 (IBM),[1] and the MOS registers for some desk calculators (Autonetics).[2]

In tracing the developments of the past by means of a few specific examples and then deriving broad generalizations, the paper ignores past mistakes to look at the optimum situation of realistic promises. Even there, however, one must compare the conflicting views on state of the art in components, such as LSI, to that in systems, such as computers.

The system engineer's state of the art is a system that has not been made before, and that may use both new and old components—though the new components must work in that system, should be reliable, and ought even to be in modest production; that is, they should have had some minimum of prior seasoning in development, application, manufacture, and use. In contrast, the component engineer's state of the art is a building block never before accomplished, carrying with it the suggestion of new functional capbailities and applications. Clearly, these two concepts of the state of the art differ radically from each other, causing misunderstanding even with realistic predictions. In the case of extravagant promises, these may later be considered misrepresentations if obstacles delay their realization as devices and in system use.

Nevertheless, these views can be brought into harmony with each other. The paper explains how each new concept, material, and component requires a certain time for development, application, and seasoning before it is ready for a new system. Furthermore, any new computer system using seasoned new

parts and components must itself be developed and tested before introduction to the marketplace as a reliable new computer rather than as an infant prodigy. The question of the immediate realization of the promises of LSI—not only in prototype components but also in practical computer systems—is therefore closely related to the time delays of our conventional design sequences leading to system manufacture.

Some device manufacturers have made extravagant predictions regarding LSI without realistic qualifications. Some qualifications have been eagerly ignored or have been misinterpreted by prospective users or system manufacturers. The acceptance of LSI in computer systems depends as much on a system designer's skill, courage, and entrepreneurial ability as on the new inventions and developments of the component maker. These factors can only be wedded after a suitable "engagement and gestation" period, which for most new computers has not yet been completed. Also costs and performance factors of competing technologies must be compared and discrepancies resolved. A few examples of imminent system applications will be cited.

The paper also discusses how both LSI and computer manufacturers share a vision of the future. The former predicts a great potential for computers; the latter, while remaining skeptical about today's LSI accomplishments, also shares this expectation. Readers, then, should obtain a better understanding of the process of assimilating new components into large systems. With this knowledge, one should be able to predict technological progress without constantly feeling cheated by the amount of work required for its realization and utilization. The challenge is how to utilize the progress in technology that has led to LSI and achieve the use of LSI in more capable computer systems in the future.

## *The promise of LSI*

Large-scale integration is derived from the extremely rapid evolution of the batch fabrication technology of silicon planar transistors. When scientists learned to fabricate hundreds and thousands of transistors next to each other on a one-inch slice of silicon—like miniature postage stamps—the idea presented itself of adding cross-connections and only separating them in blocks that contain all the interacting parts of a large gate or flipflop. As this became a reality (integrated circuits are not yet ten years old) and engraving became even finer, and functions smaller, engineers found they could place hundreds of such blocks of functions on one (by now larger) slice of silicon. Again, the thought was obvious—how many of these might one be able to interconnect and leave together on one chip? Laboratory researchers continued to apply their ingenuity to provide first multiple gates, then several flipflops, then whole shift registers or adders on one minute silicon chip; and by 1966 claimed 1000 active elements—about 100 bits—on one MOS-integrated circuit.[8] Soon one saw 16-bit bipolar memory circuits containing more than 100 active elements and transistors (Figure 1).

During the same time, electronic digital computers became even more complex. During the 1950's, 1000 or more tubes, then perhaps 10,000 transistors, provided the computing power for data processors, and operated at microsecond speeds with magnetic core memories, containing similar numbers of cores and bits. By 1964 however, the number of active elements in a computer reached 100,000, and some of the largest (CDC 6600) contained ½–1 million diodes and transistors as switching devices and used even more cores in their core memories. The expectation was that, if economically feasible, one million to ten million gates and switching elements operating at nanosecond speed would be required to provide the computing power desired for the

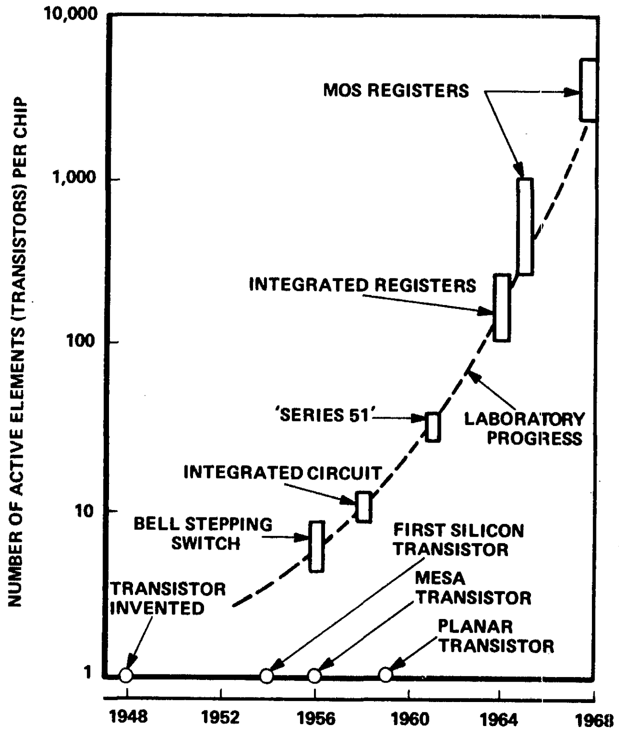


Figure 1—Complexity of integrated circuits versus year of laboratory accomplishment

largest machines[4] and their memories (Figure 2). There was much discussion on how this was to be achieved economically, practically, and without an unreasonably large effort in component assembly.

As a result, computers were ready for integrated circuits—and they are now eagerly utilizing each generation of more complex ones, as each of these in turn offers acceptable performance, ever higher speeds, lower cost per device, and greater packing density. The expectations for large-scale integration have been derived from various pronouncements made by device makers as early as 1964, and in the several succeeding years. Initially, a greater pervasiveness of integrated electronics was proposed.[5] There followed a number of laboratory investigations of complex integrated circuits and extrapolations of their characteristics were published.[6,7,8] These were quickly followed by analysis of the potential advantages of LSI from the user's standpoint,[9,10] analyses of computer organization architecture and partitioning,[11,12] as well as tempered discussions of possible areas of utilization[13] and cost.[14] Computer architecture has developed that permits interaction and utilization of large blocks of components—i.e., subsystems—without delineating all combinations of signals and their paths one by one. Thus, computer theory is capable of dealing with large-scale integrated circuits, and engineers examine all new offerings of component manufacturers to assess their suitability for one or another potential application.

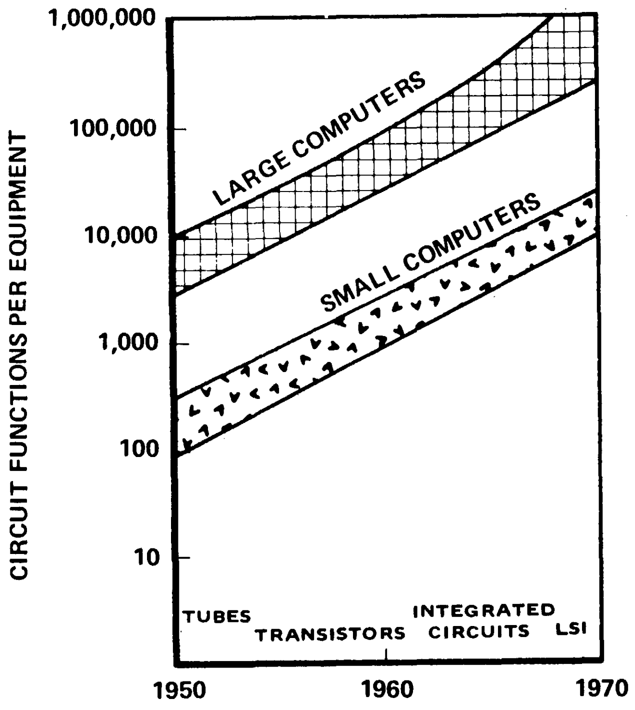


Figure 2—Functional complexity of electronic computers

On the other hand, such complex subsystems as an LSI chip must embody far more thought and care in design[15] than a simple gate circuit. The LSI chip must contain more than just a repetition and interconnection of dozens of simple integrated gate circuits. In the past, subsystems of discrete components also had to be tested, modified, remeasured, and remodified many times before they were ready for use in a large complex computer. That represents a significant change from early days, in which a transistor—if it had enough sustaining and saturation voltage, gain, and switching speed at a given cost, was considered satisfactory for a new generation of transistorized computers. At that time, it may have required one year to shake down a transistor in a new logic circuit and three to five years to develop the rest of a complex system—or two years to develop the concepts of integrated circuits, with two to four years to complete the system.[16] It might now take three years to shake down LSI ideas, and another one to three years to complete the system using them. This accounts not only for the development times required for a given product but for the total time required for developing the subsystem concepts and configurations, and adapting these to the newly conceived systems.

From such considerations various authors have derived these expectations [5–14] for LSI circuits—

- Much more complex functions—logic, memory, or other—on a single chip or a single package.
- Very low cost per elementary function or per bit.
- Far smaller size and relatively few connecting leads than present computer circuits using integrated circuits on printed circuit cards.
- Complete circuit compatibility with other semiconductor active devices.
- Off-the-shelf circuits or at least readily designed custom circuits, available with the strokes of a computer-controlled mask generator.
- A silicon device factory operated like a "Brownie" photoprint shop: put in a negative and out comes a ten-cent deckle-edged glossy print.

With such great expectations, it is not surprising that many predictions and promises were made by the device manufacturers. And many—even the more extravagant—promises were believed. Most promises made by device manufacturers were based on the concept that a further aggregation of existing standard logic or memory circuits would be sufficient to fulfill such promises. Little did they anticipate that much new technology had to be developed in order to fulfill simultaneously all or most of the above expectations on cost, ease of design, compact packaging, and so on which had been individually predicted and promised. Furthermore, computer makers have scaled up their demands and expectations, and are attempting to clarify the technical and interface requirements on purchased subsystems—and a subsystem is what LSI circuits really are.

Yet the makers of peripheral equipment, like displays or desk calculators, or of small memory buffers, are close to the realization of such promises, and are probably within a year of producing the equipment based on the expectations for LSI circuits and the promises of their vendors. The interaction between vendor and user of LSI circuits is less time-consuming for peripheral equipment systems which are much less complex than large computers.

*The accomplishments of LSI*

### Devices

At this point, it may prove instructive to look at some of the accomplishments of the semiconductor industry in more detail, from the invention of the transistor in 1948 to the complex circuits of the present time.[17–22]

Table I illustrates some key events in the steady progression of innovations utilized by the computer industry. The table shows not only the date of the laboratory announcement, but also the time (one to three years later) when such devices became available for purchase. Figure 3 presents this data in graphic form, plotting the circuit's complexity as a function of time. Note that, in addition to a delay in moving from the laboratory into first production, the mass production of silicon transistors really followed only after the planar process provided commercially useful devices at costs competitive with germanium transistors. This happened after 1960.

Figure 3 also shows the complexity of the integrated circuits actually used in computers as a function of the system's introduction date—another year or two after, production of such circuits was in full swing and produced reliable units at reasonable cost. The horizontal spread in years between these curves is a measure of the time required—again and again, one might add—to turn new concepts from the laboratory into

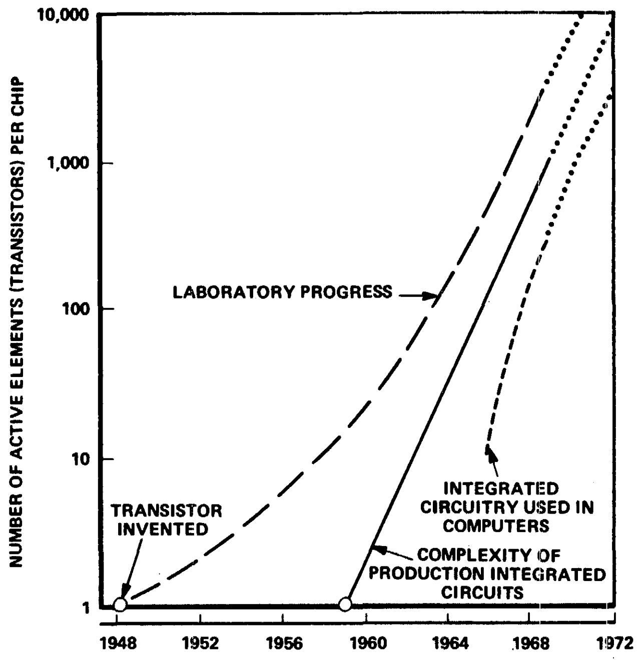


Figure 3—Twenty-year growth of complexity toward LSI

producible devices, and finally into reliable devices manufactured in large numbers at low cost.

The time span also indicates the time required for systems manufacturers to become acquainted with the properties of such devices, utilize them in prototype designs, buy a few, and again a few more; and finally to purchase many more as their systems are sold. One must remember that a device reaches large-scale, low-cost manufacture only when the system for which it is destined is also sold in large numbers.

To illustrate how many innovations must be accomplished in translating a concept into a finished device and a manufactured integrated circuit, one can look at some of the key technical innovations[23,24] and developments which led to the Minuteman II system in 1966 (Figure 4). This system employed integrated circuits in its guidance computer.

### Systems

Let us now consider several computer systems[25] that first utilized various new semiconductor devices (Table II). The years 1951-1952, when the transistor had already been in existence for three or four years, saw the advent of some of the first electronic computers using vacuum tubes. The first commercial computers with germanium transistors were introduced in 1956, when the silicon diffusion techniques were just being announced by Bell Laboratories. Diffused silicon

TABLE I—Dates of announcement of devices and circuits

| DEVICE | DATE OF ANNOUNCEMENT LABORATORY | FOR SALE |
|---|---|---|
| Transistor discovery | 1948 | |
| Germanium transistor | 1951 | 1952 |
| Grown silicon transistor | | 1954 |
| Diffused silicon transistor | 1956 | 1957 |
| PNPN stepping switch | 1956 | |
| Planar silicon transistor | 1958 | 1959 |
| Integrated circuits | 1958 | 1961 |
| MOS registers (100 bit) | 1966 | 1968 |
| Bipolar memory array (64 bit) | 1968 | 1969 |

TABLE II—Computer active devices and dates of first system shipment

| YEAR FIRST PRODUCED | SYSTEM | DEVICE TYPE |
|---|---|---|
| 1951 | Univac I | Tube |
| 1953 | IBM 701 | Tube |
| 1956 | Univac 1101 | Germanium transistor |
| 1962 | Telstar I | Silicon transistor |
| 1963 | M:nuteman I | S:licon transistor |
| 1965 | IBM 360 | SLT hybrid (silicon) |
| 1966 | Minuteman II | Integrated circuit |
| 1966 | Univac, RCA, etc. | Integrated circuit |
| 1968 | Various | MSI scratch pad memory |
| 1969 | Calculators | MOS-LSI reg'sters |
| 1969 | IBM | LSI buffer (CACHE) |
| 1970 | Various | LSI memory |

transistors did not find their way into computers until about 1963 with Minuteman I, and 1965 with the IBM SLT hybrids in the commercial Model 360. Monolithic integrated circuits did not appear until 1966-1967 in military systems (Minuteman II) and commercial computers (RCA, Honeywell, UNIVAC, Burroughs). The first large computers that will incorporate LSI are still on the drawing boards, and are expected to emerge in the early 1970's.

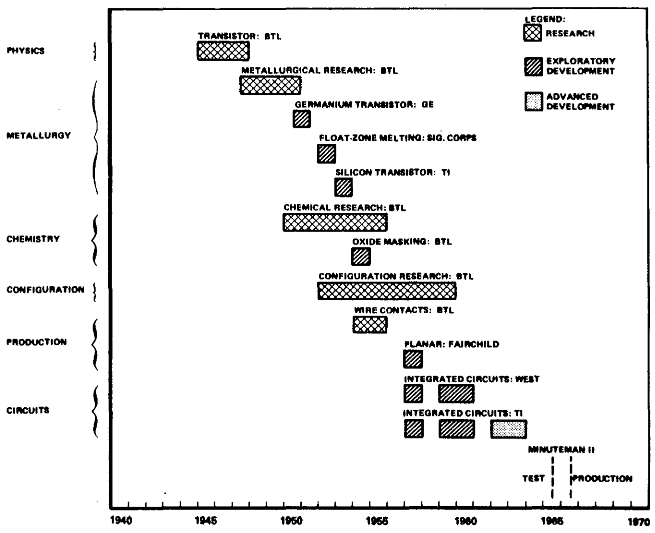


Figure 4—From transistor to Minuteman II, a twenty-year sequence of innovations in solid-state devices

## The relationship between component and system innovations

One of the reasons why systems do not immediately adopt a revolutionary concept is that the concept must have not only promise for the future, it must also compete in cost or performance with existing technologies in practical applications. Consequently, except for military applications that value lightweight or other factors of engineering performance more than cost, the germanium transistor was used in commercial computers only after it provided both higher speed and a lower cost than vacuum tubes.

The same principle holds for each later development In fact, integrated circuits exceeded the frequency. performance and cost less than most discrete silicon or germanium transistors only after 1965, and thus were not applied to commercial computers until about that time (Figure 5).

The same applies to LSI; most types described or available today are not yet out of the laboratory or are only in pilot production.[22–29] These just about match the costs of more conventional MSI or low-cost integrated circuits. Vigorous competition is not yet apparent, though it is anticipated.

Figure 6 traces the path of a system's components to some of its subsystems and systems, relating the previous data on the development dates of components and systems. For additional perspective, we have traced a few initial pertinent developments in materials and basic research.

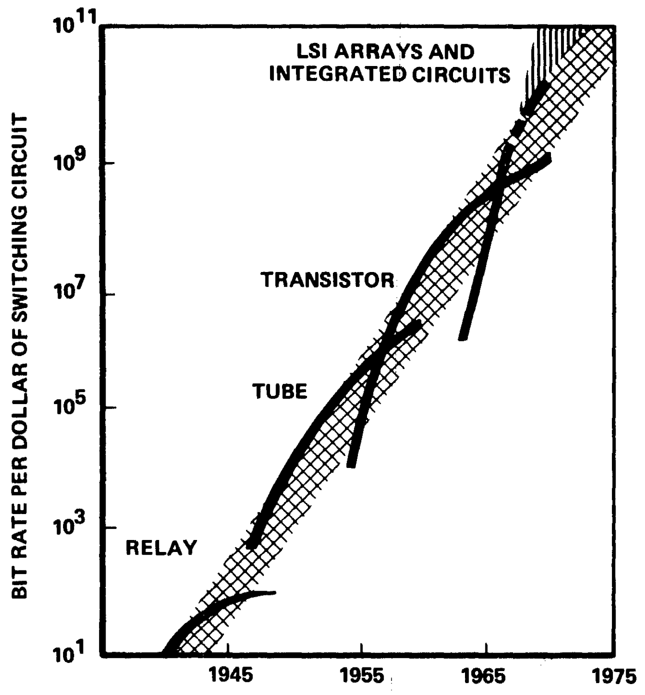


Figure 5—Switching rate per dollar for computer logic elements

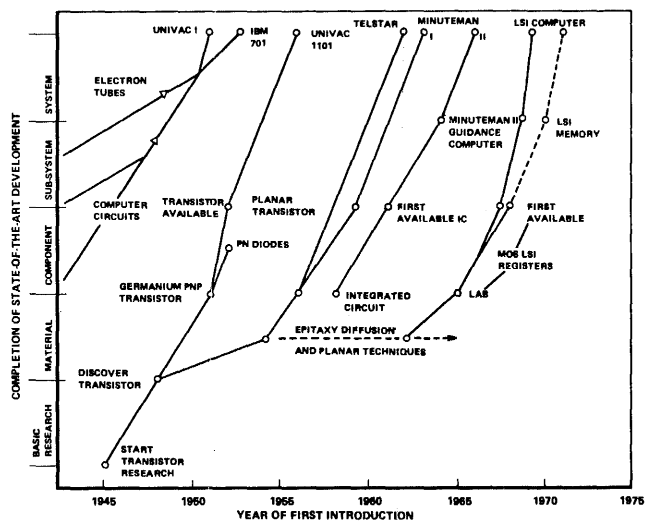


Figure 6—Tracing the development of new components into systems

*Interpretation*

One can examine[6,8,11,12] what must be accomplished in order to turn an assemblage of integrated circuits into a useful series of computer subsystems, whether logic, memory, or other. For example:

- Improvements in LSI Manufacturing
  - smaller devices with finer mechanical and optical accuracies
  - greater processing yields and lower costs
  - new package developments
  - multiple layer metallization and interconnections
- Improvements in Design
  - computer-aided logic and circuit design and tolerancing
  - automatic mask generation
  - test sequence and operation by computer
- Improvements in Applications and Development
  - diagnostic routines and their automature
  - simulation
  - development of more appropriate architecture and hierarchies for systems and memories
  - improved methods for reliability assessment

It is apparent that much of the implementation in development of such LSI circuits borrows heavily from the computer field itself in terms of mechanizing the performance of engineering design, development, test and diagnosis at many levels of device circuit, and subsystem engineering. This is in addition to the process improvements required in manufacturing the circuits.

In the new medium of the silicon crystal, one cannot test, trouble-shoot, and correct breadboards in the traditional way—that is, by using an oscilloscope or meter and test probes. The circuits are too minute, too buried under other connections and insulating layers, for po'nt-by-point signal tracing to be effective. Thus, both systems and device engineers must use new methods of diagnosis and analysis, must develop software and simulation techniques in order to understand what is going on within their own devices. This is clearly an age of computers building computers. The needs of the LSI laboratories in the semiconductor industry regarding computer design, simulation, and test make this very clear.

The device maker and the computer builder are inevitably linked to one another. In fact, the device maker might turn to the computer builder and say, "We thought you already knew how to design, test, and diagnose logic and memory circuits by use of computers. But now we find that we have to learn this from the beginning."

Even with more rapid and effective utilization of computers in LSI design, manufacture, test, and improvement, time delays must be expected between the first versions of this new concept and its becoming a reliable low-cost product, and between this intermediate step and the ultimate utilization in a large commercial electronic system such as a computer. Many interfaces must be matched, the previous, but still advancing, technologies must be overtaken, economic trade-offs performed, and investment decisions reached. Financial decisions are generally the most important, and these frequently require the longest to resolve in large organizations. Confidence in the new LSI product must be established, its reliability examined, the credibility of its manufacturer and his delivery and cost promises examined, and finally any alternative approaches again compared.

Of course this all takes time. But therein lies a dilemma. A new product will not get off the ground if someone does not risk using it; its manufacture will not be initiated if there are not at least prospective customers, and establishing reliability is difficult and expensive without prototype system use and field testing. Consequently, it is tempting to brush away the dilemma by early promises and premature announcements. Many observers believe that without forward looking claims such new concepts and developments would only evolve at a snail's pace. "Nothing ventured, nothing gained" certainly applies in this case. And the only valid realization of the promise of LSI is the delivery of such circuits and their successful use in an electronic system.

In interpreting the accomplishments to date, and the reasons why some expectations have not been realized, one discovers the following:

- The definition and structure of an LSI computer are not fully understood, but are still evolving. Yet progress toward large-scale integration appears inevitable. The semiconductor industry has a tremendous commitment and momentum toward further integration of circuits.
- Considerable time is required for the exchange of ideas and their assimilation, in order to accomplish the experimental interaction required to turn concepts into practical embodiments in systems and to test these in the field.
- While both component and computer industries may be learning from previous difficulties, many of the interactions required now between system and device designers remind one of a similar mismatch of expectations and performance requiring further interactions[30] during the early days of simple integrated circuits.

*Further expectations for LSI*

Some might conclude that the next step inevitably leads to the further integration of LSI—integration cubed or GSI (for Grand-Scale Integration). More likely, however, the evolutionary process will approach in various ways the concepts of molecular electronics, in which simple as well as extremely complex electronic functions are delineated and designed into the molecular arrangements of solids, such as a small chip of a crystal of silicon. Furthermore, it seems that the concept applied over and over is that of batch fabrication, applied to a medium particularly well suited to this concept.

The computer-on-a-slice concept may not soon be here; instead the memory-on-a-slice, the arithmetic processor-on-a-slice, the internal communication system-on-a-slice, or whatever, will be. The most likely subsystems amenable to implementation in LSI will be those suitable to repetitive batch processing, and those requiring relatively few connections to interface with other parts of the system. Significant in all cases is the repetitiveness of internal structure and partitioning that provides great functional capability with relatively few external leads. When there are a million devices inside one LSI chip, such will be called a "mega-electronic" device. But this version is still far into the future. Less complex circuits now provide good performance at low cost, and so will continue to be used for some time to come. But the underlying assumptions—that by shrinking device size further one will gain both more devices per unit area and higher internal speed—are real and lead to the expectations of still further increased performance at lower cost per function.

While many practical difficulties must still be overcome, the fundamental physical limits[3] permit at least another order-of-magnitude improvement over the performance-to-cost ratios of many present integrated circuits of medium and large scale.

Near-term applications most likely for LSI circuits are the following.[31,32]

### Memory buffers

The rapid increase in speed of logic circuits has forced modest progress in core memory speed and cost, but has far outstripped improvements in the speed of access of disc memories. Thus, opportunities for buffers between disc and core memories, and between core memories and fast logic circuits, exist. The LSI (MOS and bipolar) circuits are well suited to these respective applications, and are now being tried aggressively by some designers.

### Small memories and logic systems

The ease of interfacing with related integrated circuit logic makes semiconductor LSI memories very suitable and reasonably inexpensive for use in small systems. Pending applications are in desk calculators and in character generators for display. Existing medium-scale integrated (MSI) circuits are being used in lamp drivers and scratch pad 16-bit memory devices.

### Linear circuits

Just as applications for linear integrated circuits lagged behind those for digital circuits, so LSI linear circuits are expected to develop more slowly than digital ones. However, a number of quite complex circuits for TV and stereo radio have been developed by now, all of which certainly may be classed as medium-scale integration. Sophisticated operational amplifiers and active filters are also worthy of consideration.

### Other applications

Another widely used circuit of the future is likely to be a serial or parallel address encoder/decoder, which can be set by means of external connections or preset by the manufacturer. This class of circuit will be utilized in remote signaling and TV tuning, intercoms, mobile communication sets, and automobile or other command multiplexing systems. It also resembles certain address encoders/decoders used in computer circuits. While most of the cited applications have not yet been developed widely, they will require circuits ranging from four to 32 bits, which would barely be considered in the LSI class. Further applications are in digital differential analyses and other specialized calculator or function generator circuits.

## CONCLUSION

This paper has looked at some of the promises made by device developers about LSI and examined their accomplishments so far. The inescapable conclusion is that only medium-scale integration is here today. It will be another year before large-scale integration will be available, reliably manufactured, and accepted for use in critical portions of electronic computers.

It is also apparent from this paper that, in order to be applied in useful computer systems, technical innovations must undergo further adaptation to the specific systems, and vice versa. This mutual improvement and development requires human interaction and communication[33] during months or years of time. Of course, one can only predict the orderly progression of technology and its gestation with time, and progression and gestation may be speeded by new developments or delayed by unfortunate experiences.

One can certainly expect the future evolution of large-scale integrated circuits and their increased participation in electronic systems—not only in computers, memories, and peripherals, but also in telephone and industrial systems; and in automobile, appliance and entertainment consumer products. Only the time scale is unknown. These visions of LSI are on the horizon—to predict when they will draw within arm's reach is not the purpose of this paper. But once the first application has been successfully introduced, many more will follow rapidly.

## REFERENCES

1 J K AYLING R D MOORE G K TU
*A high-performance monolithic store*
ISSCC Digest of Tech Papers 12 1969 36-37

2 A B PHILLIPS
Private communication

3 R L PETRITZ
*Technological foundations and future discussions of large scale integrated electronics*
Proc FJCC Vol 29 1966 65-87

4 Air Force Systems Command
*Integrated circuits come of age*
Andrews AFB publication 1966

5 P E HAGGERTY
*Integrated electronics—A perspective*
Proc IEEE Vol 52 Dec 1964 1400-1405

6 R D LOHMAN
*LSI—The fabricator's viewpoint*
ISSCC Digest of Tech Papers Vol 10 Feb 1967 30-31

7 R L PETRITZ
*Current status of large scale integration technology*
Proc FJCC Vol 31 1967 65-86

8 J S KILBY
*Device fabrication for large scale integration*
ISSCC Digest of Tech Papers 9, 30 Feb 1966

9 M G SMITH W A NOTZ
*Large scale integration from the user's point of view*
Proc FJCC Vol 31 1967 87-94

10 G C FETH M G SMITH
*Large scale integration perspectives*
Computer Group News Nov 1968 24-32

11 H R BEELITZ S Y LEVY R J LINHARDT H S MILLER
*System architecture for large scale integration*
Proc FJCC Vol 31 1967 185-200

12 L C HOBBS
*Effects of large arrays on machine organization and hardware/software trade-offs*
Proc FJCC Vol 29 1966 89-96

13 M E CONWAY L M SPANDORFER
*A computer system designer's view of large scale integration*
Proc FJCC Vol 33 1968 835-845
14 R N NOYCE
*A look at future costs of large integrated arrays*
Proc FJCC Vol 29 12966 111-114
15 N CSERHALNI O LOWENSCHUSS B SCHAFF
*Efficient partitioning for the batch-fabricated fourth-generation computer*
Proc FJCC Vol 33 1968 857
16 E G FOUBINI
*The implications of solid-state technology on electronic systems*
ISSCC Digest of Tech Papers Vol 10 Feb 1967 29
17 J BARDEEN W H BRATTAIN
*The transistor—A semi-conductor triode*
Physical Review Vol 74 1948 230
18 W SHOCKLEY M SPARKS G K TEAL
*P-N junction transistors*
Physical Review Vol 83 1951 151
19 R L WALLACE JR W J PIETENPAL
*Some circuit properties and applications of NPN transistors*
Bell System Tech Journal Vol 30 1951 530
20 C A LEE
*A high-frequency diffused-base germanium transistor*
Bell System Tech Journal Vol 35 1956 23
21 I M ROSS
*A four-circuit silicon diffused PNPN stepper switch*
1956 Device Research Conf
22 *64-bit read-write memory cell*
Fairchild Semiconductor Preliminary Data Sheet No MML 9035 Sept 1968
23 *Patterns and problems of technical innovation in American industry*
Arthur D. Little Inc Federal ClearinghouseU S Dept of Commerce Rpt to Natl Science Foundation PB 181573 Sept 1963
24 *Management factors affecting research and exploratory development*
Arthur D Little Inc Federal Clearinghouse U S Dept of Commerce Rpt SD 235 to Director of Defense Research and Engineering AD 618321 April 1965
25 *Annual supplement of computer characteristics quarterly*
Adams Associates Inc Bedford Mass 1968
26 A F BEER K H NICHOLAS I H LEVIN
*A MOST memory using discretionary wiring*
ISSCC Digest of Tech Papers Vol 12 Feb 1969 142-143
27 R F HERLEIN A V THOMPSON
*An integrated associative memory element*
ISSCC Digest of Tech Papers Vol 12 Feb 1969 42-43
28 A RASHID
*High-speed LSI current mode-logic arrays for LIMAC*
ISSCC Digest of Tech Papers Vol 12 Feb 1969 68-69
29 H YAMAMOTO M SHIRAISHI T KWOSAWA
*A 40-NS, 144-bit N-channel MOS-IC memory*
ISSCC Digest of Tech Papers Vol 12 1969 40-41
30 J A MORTON
*The Microelectronics dilemma*
International Science and Technology Vol 55 July 1966 35-44
31 B AGUSTA
*A 64-bit planar doubled-Diffused monolithic memory chip*
ISSCC Digest of Technical Papers Vol 12 1969 38-39
32 T R FINCH
*LSI—Digital electronics*
ISSCC Digest of Tech Papers Vol 10 Feb 1967 130 32-33
33 J N SHIVE
*The properties, physics and design of semi-conductor devices*
D Van Nostrand Co Inc Princeton N J 1959 471

# What has happened to LSI—A supplier's view

*by* C. G. THORNTON

*Philco-Ford Corporation*
Blue Bell, Pennsylvania

## INTRODUCTION

Three years ago at the Solid-State Circuits Conference in Philadelphia, the concept of large-scale integration was already considered to be sufficiently far advanced as to be the main theme of the Conference, with a large number of technical papers showing beautiful colored slides of potential "products," containing several hundred transistors interconnected with two layers of metallization on a single chip of silicon. Related papers were presented at the Fall Joint Computer Conference that year, and semiconductor vendors had, for some time, been indicating the benefits that would accrue to the straightforward extension of the principles of planar integrated circuits to more complex "subsystems" on a single piece of silicon. The concept appeared to be clear—all that remained was its implementation; yet, as of the start of 1969, no major systems had been constructed with LSI and predictions of significant volume usage were still one to two years away. One can legitimately ask whether the darling of the industry a few short years ago has become the "bete noir" of today's computer industry, or whether most of the problems have been solved and we are well on the way to practical commercial utilization? This paper reviews some of the more significant problems that have required solution during the past four years, in order for LSI to now begin to play its role as a major element in new system design.

The situation can best be discussed in terms of the specific problem areas that have been encountered since 1964, in attempting to implement LSI. These include:

1. System design.
2. Product design.
3. Fabrication capability.
4. Testing.
5. Packaging.
6. Reliability.

It is the thesis of this paper that a number of specific problems existed in each of the above areas which would logically have been expected to require several years of effort in their solution. Each of these is discussed.

### *System design*

Since the functional density which can be practically obtained on a single MOS chip has led that obtainable with the bipolar approach, early LSI systems design experience was based on the use of MOS technology. Although individual MOS-LSI circuits were commercially available four years ago, sales for such devices to be used in conjunction with conventional components were very limited. It was quickly realized that it was nearly as difficult to build a cost effective computer system which partially used MOS–LSI, as it is for a person to become partially pregnant. For example, compatibility problems arose when systems were redesigned to use MOS rather than bipolar shift registers. Mixed systems were designed, only to find that by the time the cost of the interface circuitry and the clock drivers were included, it was more economical to use a larger number of smaller bipolar register elements. More significantly, attempts to partition parts of existing systems into blocks containing 100 gates or more led to excessive interconnections to the discrete IC control circuitry, and to new packages containing up to 60 leads. Chip sizes tended to be 14,000 mils$^2$ or larger, and it became a costly experience both to user and supplier to learn that such chips were at that time well beyond the state of the fabrication art. For optimal utilization of LSI, the system designer has found that he must rethink his system from scratch in terms of the new technology, he must be able to partition the system into tractable chip sizes with reasonable gate-to-pin count ratios, with considerable advanced care required at the partitioning step to insure the ability to test the resulting functions. It has also required studies, such as the LIMAC LSI[1] demonstration vehicle, the design of small calculators and the appearance of a variety of standard LSI functions to assist in shaping new design concepts. These concepts include distributed control and memory, with integral chip decoding and encoding, and the use of read-only memory subroutines, among other techniques. Just as the active device count required to perform a function went up dramatically when the designer went from the use of discrete components to integrated circuits, the systems designer has had to learn to waste LSI circuitry efficiently in order to make his system design compatible with the technology.

Given that the entire system must be redesigned and the associated expense, it is not surprising that most initial LSI equipments have been limited in scope. To attack the broader problem of designing large LSI computer systems or major product lines of peripheral systems, only a few user companies out of the entire industry initially made the total commitment required (i.e., 20 to 50 engineers with available in-house or vendor prototype device fabrication facilities). Such programs typically started three to four years ago using MOS technology, and have just this year reached a level of completion where prior system commitments can be made.

### *Product design*

Progress toward LSI may also have been impeded by the proffered viewpoint that the semiconductor vendor would supply the necessary partitioning and design capability. The semiconductor vendor suggested that he would integrate his facility upward to encompass subsystem design in much the same fashion as he had previously taken over much of the computer circuit design. On the contrary, many of the more successful total system programs today seem to be those where the vendor is supplying design rules relating to his fabrication capability, and the custom chip designs are being accomplished directly within the systems houses. In 1969, requirements have already existed for over 500 specialized custom chip designs needed by approximately a dozen users to implement prototype systems. The number of engineers required to accomplish these designs, even with a modern computer-aided design capability, far exceeds the number available in vendor companies. It would be irrational, moreover, to expect semiconductor device manufacturers with their general purpose circuit engineers to compete with major equipment houses in optimizing the partitioning and chip design in a variety of special system applications. Failure on the part of many system groups to get sufficiently involved in the design of custom LSI has slowed the rate of usage.

The main thrust of the component vendors has been to increase the breadth and complexity of their "standard product" lines, since it is only through volume production of such standard products that the ultimate lowest costs per chip will be obtained. For certain classes of circuits, the standard product approach is moving rapidly, with the development of such devices as shift registers, read-only memories, random access memories, A-to-D converters, D-to-A converters, BDA's, parallel-to-serial and serial-to-parallel converters, counters, etc., being made available.

Regardless of who designs the LSI components, the tools were simply not available to do the job until recently. As a minimum, the following are required:

### Logic simulation techniques

Techniques are required for simulating the performance of the blocks obtained by a trial system partitioning. Such simulation should include not only logic simulation, but should ideally take into account circuit delays. Some LSI systems designers have not been content to rely on computer simulation, but have constructed simulation cells, or macro versions of the subcircuits that they plan to work with, so that they can physically simulate the performance of the entire LSI chip. Such simulation techniques have been in development in a number of laboratories for several years, and several computer programs have also now been developed to attack this problem.

### Standardized design approaches

During the past three years, the cost of obtaining a few custom LSI chips from a vendor has remained remarkably constant in the range of \$25,000 to \$50,000, with several months to supply prototypes. Vendors and users have both attempted to improve the situation by using computer aids, and in some cases by using a standard cell or building block approach. A typical design approach is shown in Figure 1. The individual steps may all be performed manually or they can be accomplished by a computer operation. The numbers in the corners of the blocks give a rough indication of the priorities in terms of the development of computer techniques to replace manual methods.

It is noted that, after simulation and testing, higher priority is given to automatic mask generation than to the more complex problem of placement and routing. This stems from the need to eliminate the time consuming and error prone operation of ruby cutting as well as the need to obtain the required precision without excessively large camera reduction. Most large MOS-LSI chips to date have been accomplished with manual placement and routing, with computer placement and routing just becoming an effective tool.

The "standard cell" may vary in complexity all the way from a complete gate or flip-flop configuration to cells as small as individual transistor or line segments. The larger cells are easier to use in computer-aided design, and computer-aided placement and routing programs are more successful with this approach. Although the technique does not achieve minimum area, it has permitted major reduction in prototype design and turn–around time. The near practical development of all of these techniques has taken three to four years to accomplish, with more improvement to come.

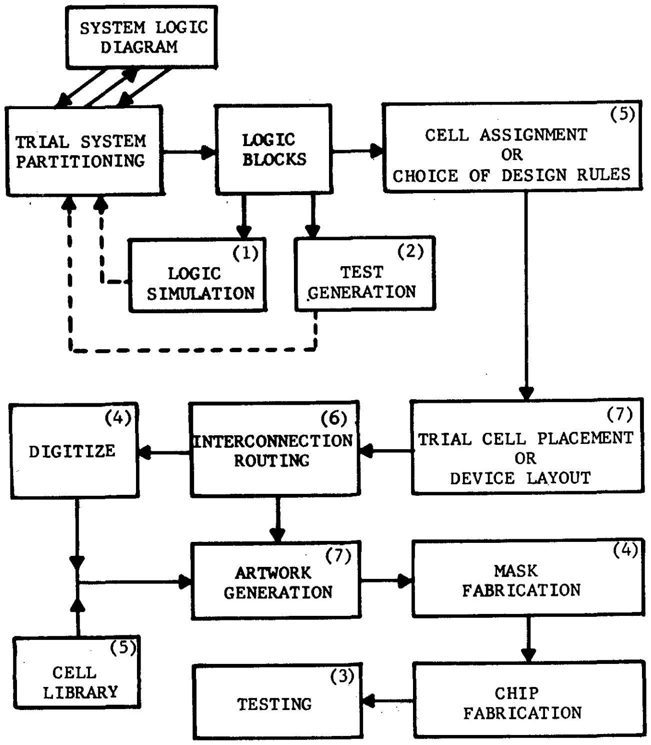


Figure 1—LSI product design

## Common design rules

Another major obstacle has resulted from the fact that multiple sourcing of user design circuits requires a certain degree of unanimity among suppliers' design rules and processes. After four years of MOS process evolution, it is only this year that parts can be ordered from as many as three suppliers, using nearly the same set of masks. The situation in bipolar has been equally chaotic, with no effective second source capability. More than one major system has gotten into serious trouble with a single source of LSI-MSI that failed to materialize. Other users are going to be very reluctant to move ahead with LSI, until some types of multiple sourcing can be found.

### *Fabrication capability*

The ease with which photographs of large complex chips with multilayer metallization could be obtained for publication a few years ago has proved to be grossly misleading in terms of the magnitude of the technical problems. As a matter of fact, a number of fundamental technical problems initially existed which made it economically impossible to produce LSI devices. Three of the more significant of these are discussed here. These are:

1. Defect density.
2. Multilayer metallization.
3. Mask making.

## Defect density

Chief among problems discussed was that of defect density, but the tendency was to greatly oversimplify the expected solution to the problem. Many managers felt that the defect density would be reduced largely by "greater care in processing," or "use of clean room facilities," rather than requiring the development and, in some cases, the invention of totally new fabrication techniques to successfully produce these devices.

In 1964, the defect problem was treated analytically by Murphy,[2] who showed that with the existing defect densities of several hundred/cm,[2] economically practical arrays could be expected to contain about 10 gates per chip on the order of 30 to 60 mils$^2$ in size. Further studies have shown that even with appreciable clustering of defects, a 98 percent yield of single gates is required to obtain a reasonable yield at the 100 gate/circuit level.[3] One approach to finessing the problem was through the discretionary wiring approach. Unfortunately, this technique developed its own set of problems, which took twice as long to solve as originally estimated. The problem of eliminating the defects was also greatly underestimated by single chip LSI suppliers, and large chip yield forecasts were made which could not be met. The sources of defects were subtle in nature, and their solution has required chemical and metallurgical process changes in wafer preparation, photoengraving, metallization, and mask making. It has only been during the past year that, under laboratory conditions and with several new process innovations, the required low defect densities (less than 10/cm$^2$) have been attained to permit fabrication of bipolar arrays containing hundreds of components on large chips. A good yield, circa 1969 (greater than 20 percent), is illustrated in the wafer map shown in Figure 2 for 256-bit shift registers, each containing 2067 transistors on 100$\times$100 mil chips.

The defect problem was thought to be simpler with MOS, in view of the smaller number of processing steps. MOS arrays did, in fact, initially yield better in somewhat larger chip sizes than bipolar, with considerably higher yields on a per-component basis because the active devices require less area than bipolar devices. However, the MOS limit was soon reached

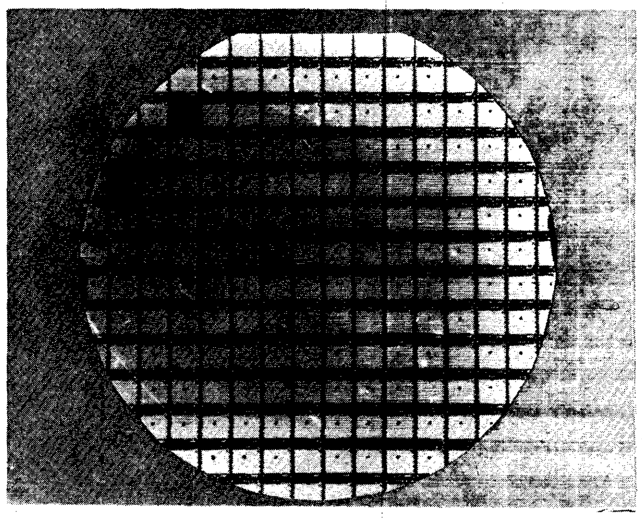


Figure 2—Map of wafer of 256-bit shift registers

at less than twice the chip size of bipolar, as it was found that each MOS process step was more critical than its bipolar counterpart. Specific MOS problems relate to the surface-sensitive nature of the devices, to the high fields which exist, and to the susceptibility of the thin gate oxide to contain specific types of defects.

Many 1964–65 MOS circuits were fabricated with only 1000 Å of oxide in the gate region. The thin oxide was required in order to overcome the high level of fixed charge density, $Q_{ss}$, in the oxide, and obtain tractable levels of threshold voltage. Clock voltages in the range of 25 to 30 V were used to overcome the high threshold voltage characteristics of these devices, and obtain reasonable speeds. Thus, fields as high as $3\times10^6$ V/cm were impressed across the oxide, with even higher yields at any thin spots that might be process induced. If one examines the detailed topology of MOS integrated circuits, one also finds stepped regions in the oxide and metal edges where even higher field concentrations exist, where defect-free devices break down when overstressed. The maximum oxide breakdown field for near perfect planar metal-$SiO_2$-silicon structures has been determined in this and other laboratories[4] to be approximately $10^7$ V/cm, not allowing for thin spots in the oxide. Thus, these devices were extremely marginal in design. It remained for the industry to learn how to reduce and control the oxide charge, permitting thicker gate oxides to be used with greater safety margins.

In addition to the problem of leakage through the oxide, MOS device performance and stability depends on the control of a number of interface effects at the dielectric semiconductor interface, most of which have, during the past four years, become well understood by physicists working in research and development laboratories, but whose control at the production level is only now becoming a reality. An example of the type of problem is that of field inversion where MOS devices lose their inherent isolation properties when the interface state density or field charge in the field oxide are allowed to vary. Specifically, in 1964, there were only a few effects associated with planar oxides that were of much concern to integrated circuit manufacturers. These included surface recombination which affected transistor $\beta$ and diode leakage, and the presence or absence of surface contaminants on the oxide which were believed responsible for the occasional channelling problems on life tests. The high doping density and low voltages used in most bipolar circuitry made these devices relatively resistant to surface problems. With the advent of MOS devices, a number of additional effects became important, and new discoveries were

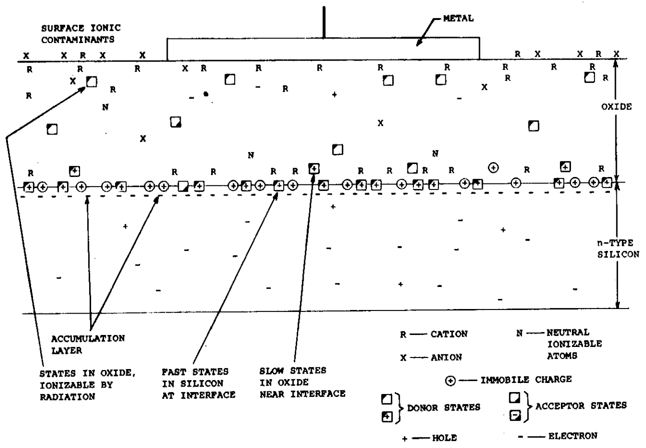

SURFACE IONIC CONTAMINANTS
METAL
OXIDE
n-TYPE SILICON
ACCUMULATION LAYER
STATES IN OXIDE, IONIZABLE BY RADIATION
FAST STATES IN SILICON AT INTERFACE
SLOW STATES IN OXIDE NEAR INTERFACE
R — CATION
X — ANION
N — NEUTRAL IONIZABLE ATOMS
⊕ — IMMOBILE CHARGE
DONOR STATES
ACCEPTOR STATES
+ — HOLE
- — ELECTRON

Figure 3—Distribution of charges in a MOS structure

made which have now been determined to affect the yield of both MOS devices and the smaller geometry bipolar devices desired for LSI. These include the presence of fast and slow ions diffusion in the oxide, the presence of fixed charges in the oxide whose magnitude is a function of processing conditions and applied fields, and a number of different kinds of minority carrier trapping effects in the oxide and at the interface. The complexity of the problem is seen in Figure 3, which shows the location of charges in a planar oxide structure.

High yield production of LSI devices requires special tests at each manufacturing step to control the important oxide charge effects. Control charts in new areas must be maintained, and the effects of process variability on these effects must be well understood by production engineering. Whereas such control has been readily understood and applied in the R&D line and at the pilot line level, many companies have been slow to implement these procedures in production, due to the considerable re-education process that is required. More than one company has been severely disappointed in their attempts to place LSI in production.

Another important limitation in increasing the yield and reliability of LSI devices has been the fact that these very complex structures literally defy analysis of internal yield and reliability problems as a function of the terminal parameters of the finished device. Traditionally, single transistors had been incorporated on each chip as an aid in process control, and for determining causes of low yield.

As the detailed nature of the many sources of device problems has evolved, it has been necessary to devise special test structures, each used to examine a particular effect in the absence of other effects. Two structures which are used for this purpose are illustrated in Figures 4(a) and 4(b). The structures shown test for the following individual effects:

1. Transistor properties and field inversion,
2. Mobile and fixed charge in the oxide,
3. Fast and slow interface states,
4. Surface ion migration and surface conductivity,
5. Leakage between p regions and leakage in large and small periphery p-n junctions under a variety of oxide thicknesses and metal overlayers,
6. Shorts and leakage through different thicknesses of oxides over different suface conductivity types and with varying topologies (small and large oxide steps),
7. Metal and p-region resistance and electromigration susceptibility under various localized conditions,
8. Metal continuity over steps,

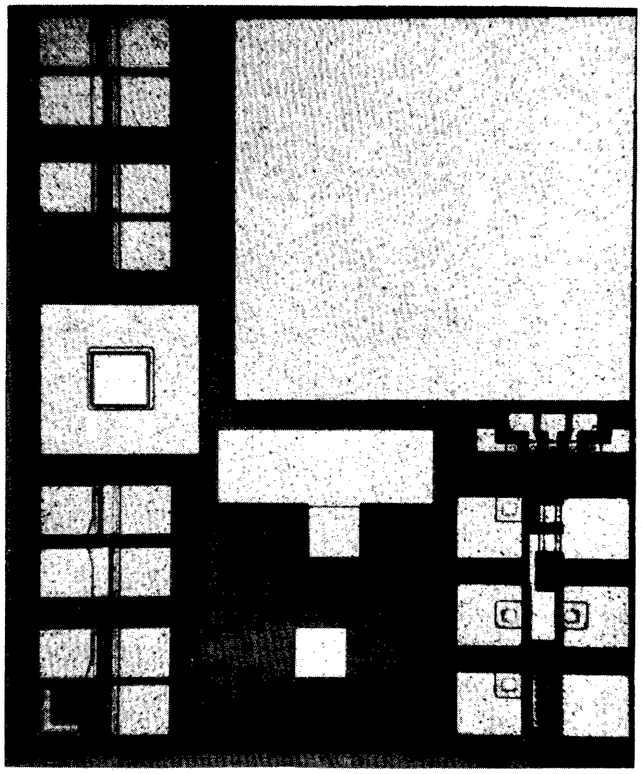
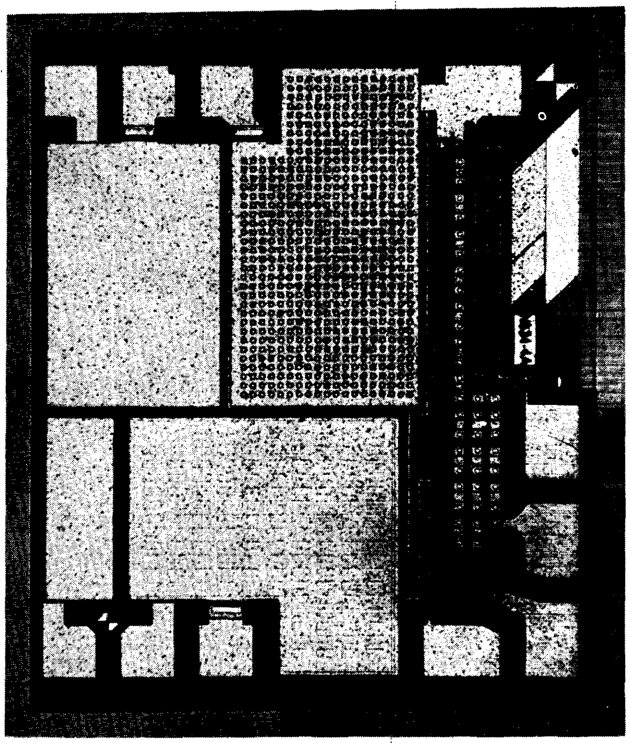


Figure 4—Test vehicles
a. Surface effects test vehicle



b. Oxide integrity and metallization test vehicle

9. Contact resistance,
10. Resistance of multilayer vias,
11. Leakage through multilayer dielectric,
12. First - and second-level metal resistance.

Such test vehicles must be used in the laboratory, pilot production, and production operations to control the process and optimize the yield, yet they took two years to develop and apply after the basic effects were known.

## Multilayer metallization

Most of the early LSI demonstration photographs showed multilayer metallization. In many cases, it is possible to obtain a 2:1 reduction in chip area with the application of an additional layer of interconnections. In the case of the discretionary wiring approach, its use was absolutely essential. Initially, it had been expected that the major problem in the use of two-layer metal would be due to shorts through the dielectric. This did turn out to be a very significant problem in the case of discretionary wiring, where an entire wafer is covered with second - and third-level insulated interconnections which must be free of shorts.

In the case where the actual shorts through the oxide are not present, there may still be a large number of thin or weak spots which are susceptible to premature breakdown. The evolution of a uniform high-strength dielectric for multilayer technology involved tests such as those shown in Figures 5(a) and 5(b). In this type of test, thin metal is used in the upper layer so that when a short develops, the energy dissipated will evaporate the metal away from holes—thereby "clearing" the short, and restoring the original condition. Thus, it is possible to impress consecutively higher and higher voltages between the two layers, exposing the weak spots one-by-one, until the ultimate

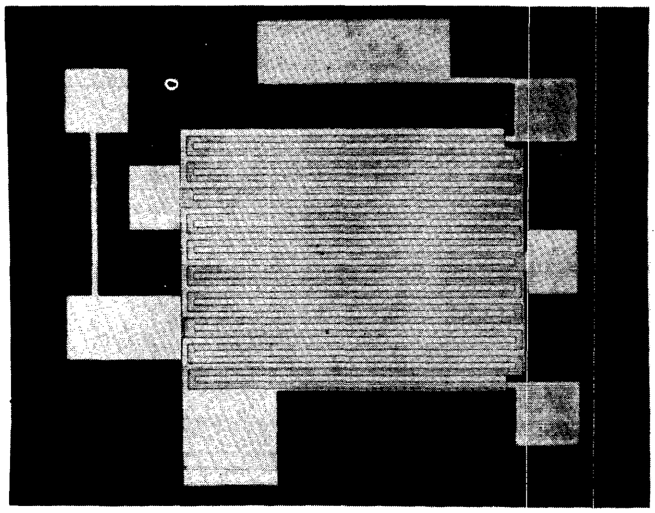
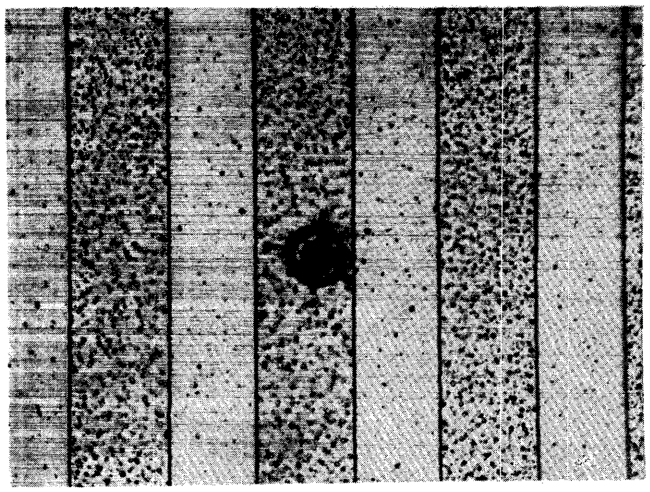
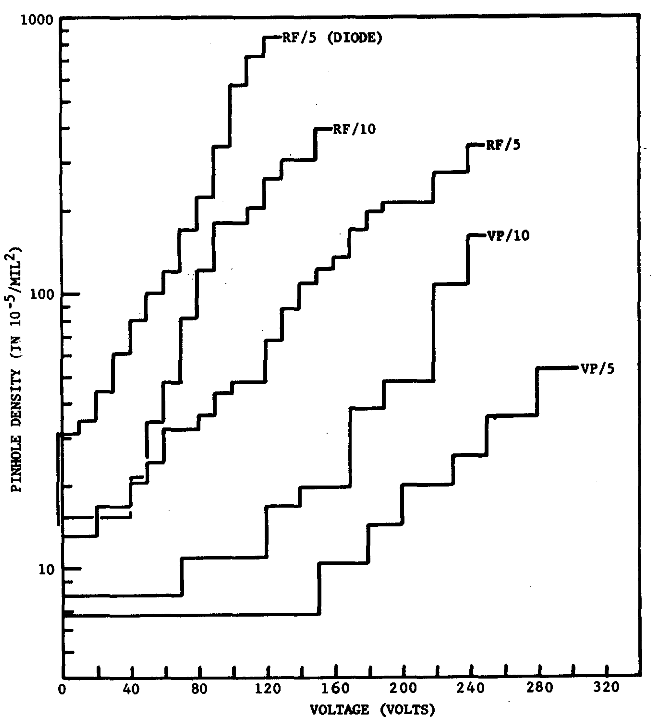


Figure 5—Breakdown strength of oxides in a multilayer test vehicle
a-1. Dielectric strength test vehicle



a-2. Enlargement showing self-healed pinhole



b. Pinhole density for silane-vapor-plated and R-F sputtered $SiO_2$ on delineated aluminum 5,000 Å and 10,000 Å thick

dielectric strength is determined. The "stair-step" plots, shown in Figure 5(b), show the wide differences between silicon oxide dielectric layers prepared with differing processing conditions. It is now possible to prepare both chemical vapor-deposited and sputtered $SiO_2$ layers which are virtually free of shorting-type defects within the area of a single LSI chip, and success is also being reported on large discretionary wired wafers with a combination of these techniques.

In the case of smaller chips, opens proved to be of more significance than shorts, with problems developing at the vias between upper and lower metallization levels. In order to limit their size, such vias must be kept small in area, and it was quickly determined that the presence of thin oxide layers or other contaminants at these points would produce either opens or an unacceptable amount of via resistance. Under non-ideal conditions, a test structure such as included in Figure 4(b), containing 18 vias in series, commonly shows resistances on the order of 10 to 20 ohms. In some LSI circuits, the tendency for a high resistance to be present is increased by the occurrence of cell potentials, which produce an anodizing effect during via etching, and which is a function of a particular circuit topology. Thus, the same four vias in a circuit containing 120 vias might be found to be open without any obvious reason. Metallization problems also developed with electrical opens occurring at the point where the upper-level metal steps down over an abrupt oxide cut to reach the first-level metal, and the metal at these points tends to become constricted. It appears to have taken the better part of two years of effort in various industry laboratories to develop multilayer processes to the point where they can be used to achieve competitive yield and reliability levels with single-layer metal products. Even so, rules governing the via area znd shape of the via cut must be carefully chosen and strictly adhered to.

The application of multilayer metallization to MOS is less critical for via resistance, since the circuit operate at high impedance levels. A different type of fundamental problem arose, however, when it was found that the application of the second layer of dielectric caused drastic changes in the electronic properties of the first-level silicon-oxide interface.

Not only temperature and radiation effects (in the case of sputtering processes) exist, but rapid diffusion impurities can be introduced which penetrate to the original interface and alter the charge condition. Thus, the same level of new understanding and special process control is required as was the case in the original development of stable high performance MOS devices.

## Mask making

In 1964 and 1965, severe problems existed in mask making which alone would have made it impossible to manufacture LSI. Problems existed in both image quality and image registration.

In the case of image quality, lenses were not generally available to handle the conventional 10X final step-and-repeat reduction with a sharp field in an area greater than a 75×75 mil chip. Attempts to step at a larger size and then reduce a multiple pattern were also limited by the lens quality and photoprocessing techniques, so that considerable size and corner compensation had to be built into the original artwork to obtain something close to a usable mask.

As better lenses became available, image quality improved, but problems remained in sizing and registration which still limit maximum practical array size. A high yield of circuits of typical "state-of-the-art" design generally requires the placement of successive images, one within the other, with a separation of a tenth of a mil, and a tolerance of this of 0.05 mil. In an LSI device, one might logically wish to obtain such registration at opposite ends of the diagonal of a 115 mils$^2$ chip. In the mask-making stepping process alone, three sources of error occur (under optimum conditions) which affect this registration: (1) vertical stepping error $\pm 0.01$ mil, (2) rotational stepping error $\pm 0.01$ mil, (3) size reduction error $\pm 0.02$ mil (1 part in 4000 reduction error over a 2″ stepping table travel). Adding these tolerances leads to $\pm 0.040$-mil registration error in the mask, which means that the processing operator must align her mask during device fabrication to $\pm 0.01$-mil—a bare possibility. Thus, any attempt to fabricate circuits at sizes larger than 115 mils on a side with 0.010 mil registration requirements has automatically placed severe limitations on the expected yield, and practical bipolar LSI design rules have therefore been kept to larger tolerances or smaller chip sizes. Unfortunately, optimum MOS performance demands even tighter design rule tolerance ($\pm 0.08$ mil on gate overlap).

*Testing*

LSI raised many new problems in testing, some of which were initially recognized and some which only became evident when manufacturers attempted to move LSI testing to the production level. It is now generally recognized that for circuits containing more than 50 gates, one cannot practically exercise all of the logic contained on the chip as a method of testing, since the time required to accomplish this quickly stretches into many hours or days per circuit; rather, test programs must be computer-generated which rely on the fact that only certain kinds of faults can practically exist in the device, and which merge redundant test patterns. Some fault conditions can only be detected by the introduction of specially constructed error inputs. Even with such factors taken into account, however, an effective test sequence can only be expected to become available when the test problem has been taken into account at the time of system partitioning and circuit design. In some cases, it is necessary to break feedback lines on a chip to reduce sequential networks to combinational networks, albeit at a sacrifice of gate-to-pin ratio.

At best, a formidable problem still presents itself. Two of the major contributors to this problem are: first, the inability to test the "inner stages" of the array, resulting in an inordinate number of tests necessary at the inputs to guarantee the proper outputs, and second, the complex test sequence generally exceeds the capability of available test equipment and might be expected to add a disproportionate amount to the total cost of the device. As the level of integration increases, the number of actual chips per system will decrease, but the cost of testing fewer (but more complex) chips can become the most significant contributor to the final cost of the unit.

The testing of sequential logic can be considerably complicated by the necessity of first applying a sequence of input patterns to force the output into a particular state. Consequently, consideration must be given to the sequence of the input patterns to ensure a complete functional test. As with combination networks, a test pattern for sequential networks can be reduced through the use of computer-aided test minimization programs. However, these programs can be quite long, hence, expensive, since many distinguishing sequences are necessary to check the possible failure modes.

Although the problem of generating sufficient test programs has in many cases been satisfactorily resolved, the problem of how the testing is to be performed on a manufacturing basis is still largely undecided. As of the New York IEEE Show in March 1969, for example, only two or three pieces of commercial equipment were being offered for LSI testing, and in general these equipments are either considerably limited in capability or are very expensive, as applied to single operator handling. Examination of these equipments and other individual test equipments, which exist in individual companies, would suggest that we are still in the first generation of LSI test equipment development. Progress in the commercial use of LSI will continue to be impeded until this problem is resolved.

*Packaging*

LSI raised many new problems associated with packaging these devices. Early attempts at LSI system partitioning led to poor gate-to-pin ratios in an attempt to maintain maximum system flexibility which in turn required large numbers of bonds. Initially, attempts at packaging such LSI were extensions of the then available flat pack techniques attempting to maintain a minimal periphery chip with a large number of closely spaced leads. This configuration led to a shorter seal length than had been determined by the package industry to be required for freedom from leakers. The urgent necessity for having packages suitable for prototypes also led to the use of less than optimum procedures for fabricating and sealing these packages. Sealing techniques which worked well on small integrated circuit packages failed to seal properly when the package periphery became large, and special techniques had to be developed. Conventional leak test procedures cannot be applied, since the larger flat packages will not withstand the same test pressures and the larger internal volume requires excessively long pressure tests to detect small leaks. The larger number of pins also put new requirements on the wafer and chip bonding processes. Large chips are more likely to have voids in the chip-to-header bond, and a larger number of wire bonds have to be made without a bad bond in order to obtain a finished device at high yield. One solution to this problem has appeared to be in the direction of beam lead or flip-chip techniques. The applicability of such techniques to large numbers of interconnections has relatively recently been demonstrated, as in the case of the semiconductor memories described by Kraynak,[5] Agusta[6] and Alexander.[7] Most of these approaches have required additional processing steps on the wafer to obtain the required bonding materials at each interconnection site.

High speed LSI arrays have also placed new demands on packaging from a power dissipation standpoint. For example, an array of 100 high speed gates, each dissipating 50 mW, would produce a total dissipation of 5 watts, which has been beyond the state-of-the-art of conventional IC packaging. High speed LSI has therefore required considerable research into methods of obtaining high speed at lower power levels, and this has required smaller geometry structures to minimize capacitance—thereby making the large LSI circuits more difficult to produce at a reasonable yield.

### *Reliability*

One of the originally stated reasons for going to LSI has been to increase reliability by decreasing the total number of interconnections and packages in the system. This may be true for a system of fixed capability, such as a desk calculator or a computer terminal. On the other hand, in large systems, LSI is more often viewed as a means of economically increasing the total system complexity to perform more tasks, rather than as a means of decreasing the package count for previously designed systems, in which case the MTBF for the total enlarged system is of concern.

The advent of LSI brought into the picture a new range of potential reliability problems that have to be resolved.

Since LSI devices are more complex, they require more metallization per chip. The larger number of pins in LSI leads to an increased number of interfaces between the chip and the package, and it is at these locations—wire bonds and chip bonds—that the principal failure modes occur in silicon integrated circuits. In fact, metallization and wire bond failures account for approximately 60 percent of all conventional integrated circuit failures. Thus, reliability may suffer on a per package basis.

In the case of high density LSI in a conventional type of IC package, dissipation is increased to the point where the circuit elements are operating considerably closer to the maximum allowable junction temperature than would be the case for individually packaged lower complexity IC's. Derating to increase reliability is not as feasible and it has become important to explore the long term degradation of devices at these higher temperatures that can no longer be considered an accelerated condition.

Failure rates on a per package basis are necessarily increased by this effect, and the MTBF for the entire system must be re-evaluated to make sure that the expected benefits are in fact being obtained.

Perhaps the area of greatest difficulty in insuring LSI reliability is in the application of the screening techniques that have been accepted for use in integrated circuits. Typically, visual, mechanical, thermal and operational screening of the final product is required. The final in-process screens should be performed at stress levels sufficiently stringent to remove all devices which contain potential reliability hazards, but the screen levels imposed must not degrade the inherent reliability of those devices which survive the screening sequence. Unfortunately, the screening levels adapted for conventional integrated circuits, however, may not be applicable in general to LSI and MSI devices.

Because of the larger size of LSI packiages, the centrifuge and shock tests applied to conventional LSI can cause mechanical damage and loss of hermeticity unless special precautions are taken.

Because of the increased complexity of IC's and MSI devices, it must be assumed that the effective ness of a preseal visual inspection will not be as great as it is for conventional integrated circuits. The sheer complexity of these devices outstrips the ability of a human operator working with a microscope. This is particularly true when one considers the increased number of possibilities for scratches and open metallizations at oxide steps, the possibility of shorting between upper and lower metallization levels because of pinholes or cracks in the insulating oxide, the possibility for opens due to marginal metallization alignments, and the possibility of failure because of high leakage between adjacent metallization stripes because of photolith defects, resulting in poor delineation.

## SUMMARY

The promises of LSI are still basically valid; however, the electronics industry has had to face tremendous problems in its efforts to make LSI a production reality. The solution to these problems has required the development of new approaches in almost every aspect of integrated circuit technology, and has required close cooperation between the vendor and the user. It is, in fact, remarkable how much progress has been made in the past four to five years. At present, there are over 200 catalog part numbers for LSI devices and several LSI systems are programmed for some 1970 production. It now appears that 1970 will be the year of reality for LSI.

## REFERENCES

1 G HERZOG
*The LIMAC—An LSI demonstration vehicle*
IEEE International Convention Digest N Y Mar 26 1969

2 B T MURPHY
*Cost optima of monolithic integrated circuits*
Proc IEEE Vol 52 1964 1537-1545

3 A G F DINGWALL
*High yield processing for fixed-interconnect large scale arrays*
IEEE Trans on Electron Devices Vol 15 No 9 1968 631-637

4 N KLEIN H GAFNI
*The maximum dielectric strength of thin oxide films*
IEEE Trans on Electron Devices Vol 13 No 12 1966 281-289

5 P KRAYNAK P FLETCHER
*Wafer-chip assembly for large-scale integration*
IEEE Trans on Electron Devices Vol 15 No 9 1968 660-663

6 B AGUSTA
*Planar double diffused monolithic memory chip*
Digest of Technical Papers Solid-State Circuits Conference Feb 19 1969 Philadelphia Pa 38-39

7 E J ALEXANDER
*P-channel IGFET memories*
IEEE Internat Convention Digest March 26 1969 N Y

# Real-time graphic display of time-sharing system operating characteristics*

*by* JERROLD MARVIN GROCHOW**

*Massachusetts Institute of Technology****
Cambridge, Massachusetts

## INTRODUCTION

The Graphic Display Monitoring System (GDM) is an experimental monitoring facility for Multics, a general purpose time-sharing system implemented at Project MAC cooperatively with General Electric and the Bell Telephone Laboratories.[2,7] GDM allows design, systems programming, and operating staff to graphically view the dynamically changing properties of the time-sharing system. It was designed and implemented by the author to provide a medium for experimentation with the real-time observation of time-sharing system behavior. GDM has proven to be very useful both as a measuring instrument and a debugging tool and as such finds very general use.

Monitoring the activity of a traditional computer system (one with only a single active process) is a fairly simple task. Hardware and software devices can easily be devised to keep track of almost any parameter. Asking the question "What are you doing right now?" to a computer system controlling multiple processes or servicing multiple interactive users, however, proves particularly difficult to answer meaningfully. It becomes necessary to "snapshot" the system (record in some manner its state at a specific time) and interpret this information for the inquirer. Since a basic property of a time-sharing system is that, in fact, it is "doing something else" a few milliseconds from now, what the inquirer really wants to ask is "What are you doing now, and now, and now . . .?" Implicitly, he is also asking to be shown what is happening in an easily interpretable format. The GDM solution to his problem is to provide the user with a real-time, graphical output "eavesdropper."

Statistical studies of time-sharing systems have been performed[1,5,11] in an attempt to provide "after-the-fact" monitoring (in effect answering the question "On the average, what is happening?") and there have been simulations in an effort to provide "predictive monitoring."[5,11] One company has even produced a hardware device to receive system status information over a special wired in channel and record the results on magnetic tape.[12] Other than the "SNUPER Computer"[6] which, however, still requires engineer-installed hardware probes, there has been little work directed towards providing a generalized, real-time, time–sharing system monitoring device. It is felt that while the hardware used for this implementation of GDM is perhaps unusual, the design principles involved and the monitoring methods explored are sufficiently general to provide a framework and a guide for other designers.

The basic goal in designing the GDM System was to produce a time-sharing system monitoring device for use by the staff of the Multics project. Initial requirements implied that it would be on–line, that is, active while Multics was in operation—not just collecting data for future analysis, and would provide

* Work reported herein was supported (in part) by Project MAC, an M. I. T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01)

** This paper is based on a thesis submitted in partial fulfillment for the degree of Master of Science at the Massachusetts Initutest of Technology, Department of Electrical Engineering.

*** Project MAC

dynamically changing graphic output (as well as hard copy if desired). It was to be designed such that the act of monitoring did not cause significant interference to the time-sharing system or perturbations in its behavior and such that it would not be necessary to make more than a few minor additions to supervisory procedures in order to incorporate the GDM System (as opposed to monitoring done by inserting entire procedures in critical points in the supervisor in order to collect data; see Scherr[11] for an example). Since the GDM System was to be an experimental tool, it was also considered especially important that it be easily expandable and adaptable to new or different monitoring requests. Coupled with these requirements was the need to involve the expected user community as early as was possible in the project in order to insure its continued use after initial implementation. In this regard, acceptance by the systems programming staff was very encouraging and many currently make use of the GDM facility.

The original GDM System embodies these goals while making use of existing hardware at Project MAC. The Digital Equipment Corporation 338 (see Figure 2) was already on site for use in other experimental work. A more extensive (and less expensive) monitoring system could perhaps be designed if it were possible to choose both the display processor and the method of interface to the time-shared computer. This was not, however, viewed as a major handicap in developing a useful system.

Succeeding sections will discuss the various components of the GDM System and will describe in detail initial experiments and current usage at Project MAC. Compromises in design and special problems due to the particular constraints of the display hardware or software and the Multics system to which they interfaced are also discussed.

## *What is the GDM system?*

### Subsystems

The GDM System consists of four major components:

A. An input-output procedure running under Multics to transmit data as requested to the display computer.
B. A monitor system operating on the display computer to facilitate the creation, storage, and retrieval of display templates (see below) and to perform various other housekeeping functions.
C. A series of display computer subroutines for manipulating data and generating command sequences for the display.
D. A language for describing desired data manipulation and display formats (Display Description Language), a (planned) compiler for translating such descriptions into display computer assembly language programs, and a set of macro-definitions for simplifying display computer programming and for calling the subroutines mentioned under C.

Figure 1 gives a functional representation of the various GDM subsystems showing the interaction among them, the two computers, and the user. Figure 2 shows the complete hardware configuration. Reference 8 goes into considerable detail about the GDM monitor system software including system flow charts.

### Modes of operation

Use of the GDM System generally falls into one of three classes of operation:

1. Demonstration mode: any of a number of library displays may be viewed to get a general picture of Multics operation at the moment. Data used in these displays is updated periodically according to preprogrammed instructions.
2. XRAY mode (so named because of its similarity to the X–ray System[4]): the user may type the

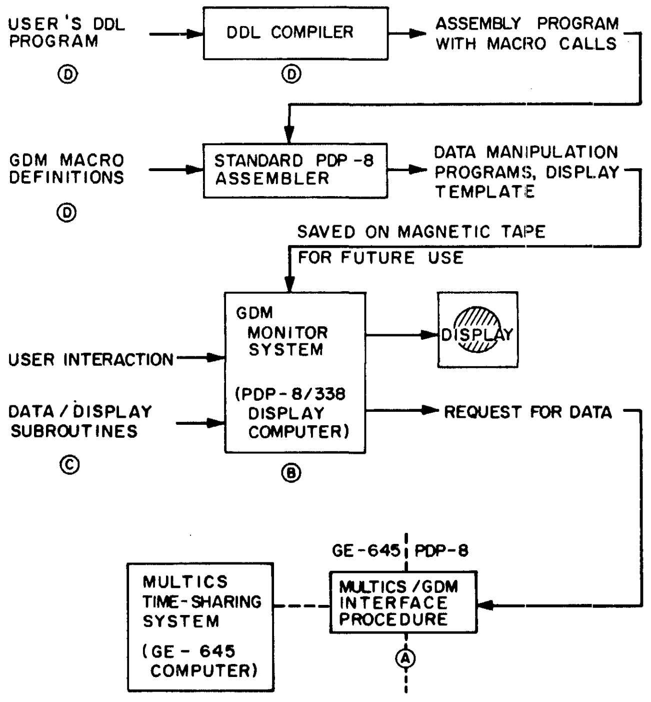

USER'S DDL PROGRAM
Ⓓ
DDL COMPILER
Ⓓ
ASSEMBLY PROGRAM WITH MACRO CALLS
GDM MACRO DEFINITIONS
Ⓓ
STANDARD PDP-8 ASSEMBLER
DATA MANIPULATION PROGRAMS, DISPLAY TEMPLATE
SAVED ON MAGNETIC TAPE FOR FUTURE USE
GDM MONITOR SYSTEM
(PDP-8/338 DISPLAY COMPUTER)
Ⓑ
DISPLAY
USER INTERACTION
DATA / DISPLAY SUBROUTINES
Ⓒ
REQUEST FOR DATA
GE-645 | PDP-8
MULTICS TIME-SHARING SYSTEM (GE-645 COMPUTER)
MULTICS/GDM INTERFACE PROCEDURE
Ⓐ

Figure 1—GDM subsystem interactions

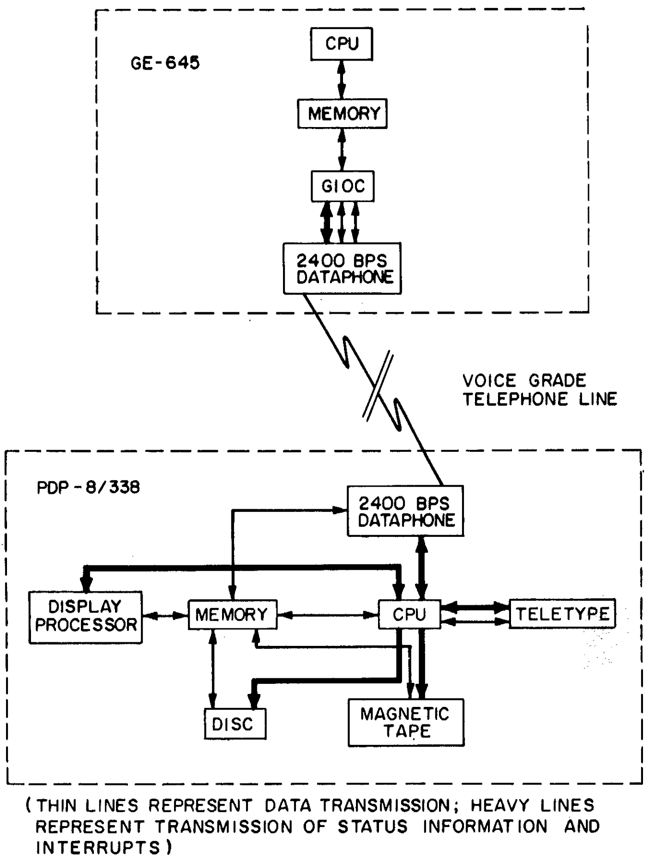
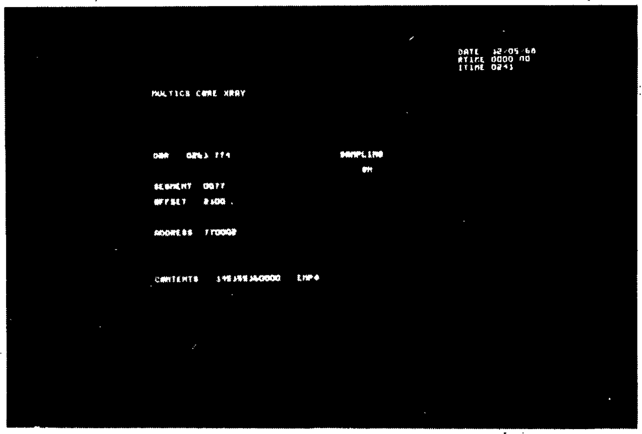

GE-645
CPU
MEMORY
GIOC
2400 BPS DATAPHONE
VOICE GRADE TELEPHONE LINE
PDP-8/338
2400 BPS DATAPHONE
DISPLAY PROCESSOR
MEMORY
CPU
TELETYPE
DISC
MAGNETIC TAPE
(THIN LINES REPRESENT DATA TRANSMISSION; HEAVY LINES REPRESENT TRANSMISSION OF STATUS INFORMATION AND INTERRUPTS)

Figure 2—Hardware configuration

segment number and offset of a datum (see Reference 3 for a description of the addressing scheme used in Multics) on the teletype of the display computer and see displayed the octal and ASCII character representation of its contents, updated every second (Figure 3, XRAY display).

3. Display creation mode: the user will go through the process of creating his own display (as outlined in Figure 1) in order to gain desired flexibility in data displayed, format of display, or data sampling rate. Displays are then saved in a special format, the "display template," for use in later experiments or as part of the library.

All modes of operation employ the same type of display template and are listed only to differentiate between the application of the GDM System. System programmers have been trained in five minutes to utilize the many displays already in the library (operation under "Demonstration Mode"). Some use the XRAY display when there are one or two locations of interest at a particular moment, as in the current



```
                                        DATE  12/05/68
                                        RTIME 0000 00
                                        ITIME 0243

MULTICS CORE XRAY


DBR     [illegible]              SAMPLING
                                    ON

SEGMENT   0077
OFFSET    2100 .

ADDRESS   [illegible]


CONTENTS   [illegible]   [illegible]
```

Figure 3—XRAY display

number of available disk pages or the value of a particular time-dependent variable. Display creation mode, the most general use of the GDM System, requires the most work on the part of the user. He must decide what data items to display, how to display them, and how often to sample them. He must then create the data manipulation routines and the display list comprising his particular "display template." Until the DDL compiler is constructed, this work must be done in an extended version of the PDP-8 Assembly Language as seen in Table I (the 338 computer uses the same systems software as its sister PDP-8). It is in this mode of use that all the facilities of the GDM System come into play and in which the most fruitful experimental work can be performed.

*Examples*

Figures 4 and 5 show typical examples of GDM displays. Figure 4, Core Memory Summary Display, displays real-time information on the usage of Multics core memory pages; Figure 5, Active Process State Display, displays user activity information (see below). The display templates for both figures were constructed in about two hours apiece by an experienced user and have provided many hours of system observation for experienced and inexperienced alike.

The display in Figure 5 causes information about each process in Multics to be extracted from the traffic controller data base. The column labelled "MP" is the "multiprogramming state," an indication of a process' eligibility to receive CPU time. Stars to the right of this column indicate the processes that are eligible (state 4). The column "ST" is the "activity state"—running, ready to run (waiting to be serviced), or not ready to run. The star is next to the process currently

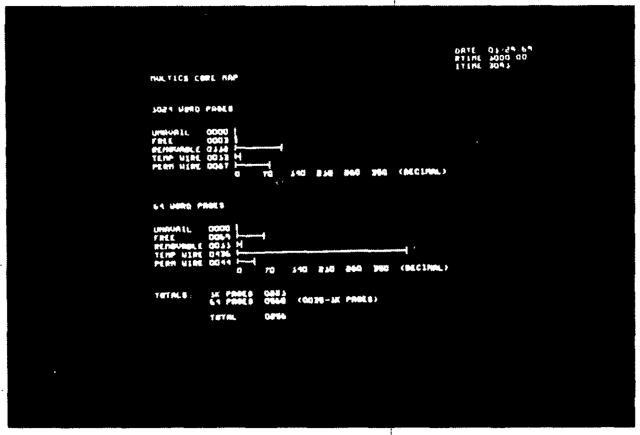


Figure 4—Multics core memory summary display

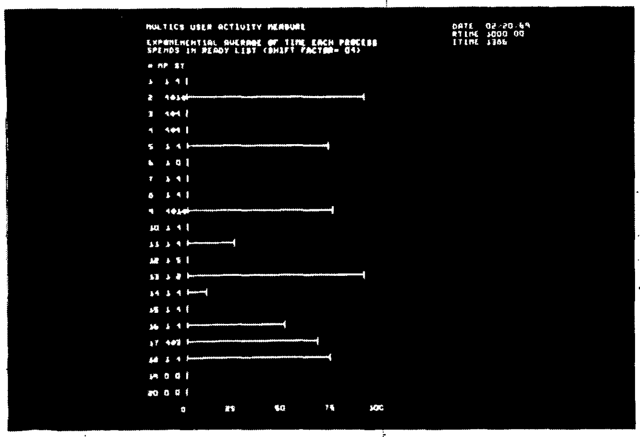

MULTICS USER ACTIVITY MEASURE
EXPONENTIAL AVERAGE OF TIME EACH PROCESS
SPENDS IN READY LIST (SHIFT FACTOR= 04)
DATE 02/20/69
RTIME 1000 00
ITIME 1306
# AP ST
0 25 50 75 100

Figure 5—Active process state display

running, state 1. In a multi-processor configuration, there would be more than one such process.

The associated bar graphs also provide a descriptive measure of overall system activity. By "eyeball integration" of the length of the bars, one can get a fairly accurate idea of system loading. Several means of calculating graph lengths have been used (in different display templates all using the same basic form):

1. Whenever a process is ready or running, the length of the bar is increased. When the process is not ready, the associated bar decreases. Each bar changes length as an exponentially weighted sum of ready-running and not-ready time. (This is seen in Figure 5.)
2. Whenever a process is ready, its bar grows in equal time increments. When the process is finally serviced (receives processor time), its bar is reset to zero length.

The display of type 1 gives a general picture of system loading but also shows something of the behavior of the individual process. The scale is calibrated in percentage to indicate that the bar shows the percentage of time a process is requesting or receiving CPU time—a measure of the process' activity. The type 2 display is more useful in getting an uncluttered picture of just how long a "ready" user must wait to run, i.e., how long each process is spending in the queues waiting for service.

The display templates for these two displays differ in about ten instructions (the computation of bar length). The two hours of editing and assembling to get a "first draft" of the display is even less if averaged over the two displays. Herein lies a basic flexibility of the GDM System: once the data to be displayed have been decided upon, it need be only a matter of minutes before it is viewed. Display formats can be easily experimented with and a finished display template can be added to the GDM library for future monitoring without any costly "dedicated system" monitoring runs.

The examples discussed above show simply two ideas. Others have included collecting (and displaying) data on the mean lifetime of a page in the Multics memory (how long does it take before the page is swapped out to secondary storage), the distribution over time of the number of active time-sharing users (very nicely displayed as a graph similar to Figure 6D), and the average number of users referencing particular supervisor segments (built up during the length of the monitoring session). There is a great deal of work yet to be done before we run out of ideas or into the limitations of the GDM system.

*More on the display template*

A display template (DT) consists of three sections:

1. A list of the time-sharing system data items to be sampled (segment number and data base format are sufficient since absolute core locations are determined by GDM at monitoring time).
2. Instructions on display type (numerical, ASCII, bar graph, other graphics, etc.), sampling rate, and data manipulations (averaging, scaling, etc.) for each data item or group of items.
3. A display list: machine instructions for the 338 Display giving text, formatting information, and storage for items to be displayed.

For example, to display a single process' activity as

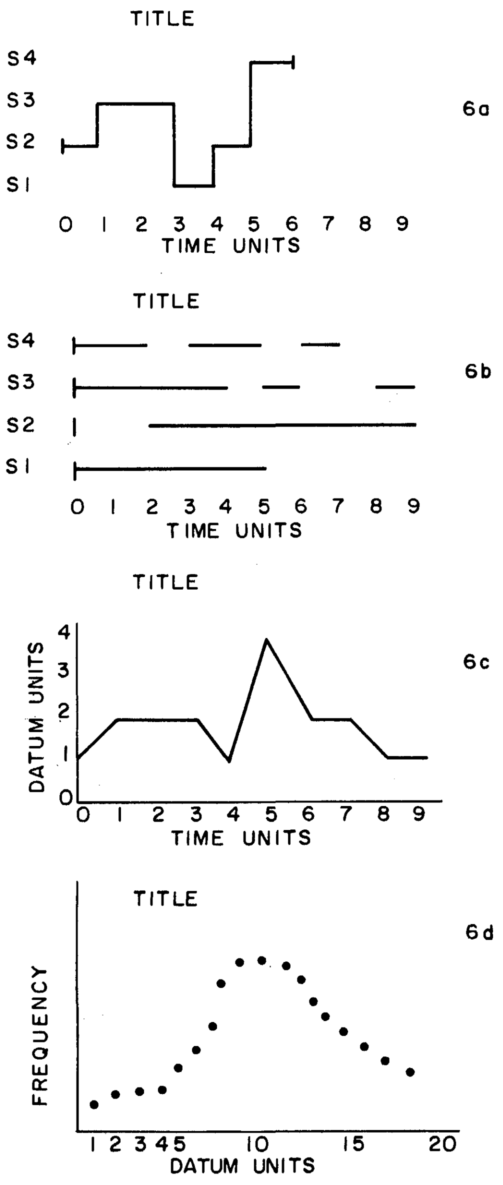


Figure 6—Other standard display types

in Figure 5, a DT would contain about twenty instructions (Table I).

The various non-PDP-8 instructions (call, do, dlstart, etc.) are macro calls to a set of definitions designed as part of the GDM System. Various subroutines (nplot, ge645, sked, etc.) are also provided as interfaces to the GDM monitor and to simplify programming. These features allow the programmer without PDP-8 experience to design a display template with a minimal apprenticeship. (Implementation of a DDL compiler should simplify this even further.) Of course, since all the facilities of the computer are available, data manipulations can be quite complex (although subroutines are provided for such common operations as scaling and masking) and displays quite unusual (Figure 6 shows standard types for which GDM provides some macro facility). The only limit is the designer's imagination and the size of the PDP-8 core memory.

### *The Multics/GDM interface*

The GDM System is designed for use in a symbiotic relationship with a time-shared computer. The computer must be capable of supporting a display processor functioning basically independently of the time-sharing system but occasionally interjecting requests for data transmission.

The Multics environment is particularly friendly to this type of system as it is possible to make data requests through the generalized input-output controller (GIOC) of the GE-645[9], without interrupting the central processing unit (Figure 2). It is necessary, however, to dedicate two of the 2048 GIOC channel pairs (one for transmitting and one for receiving) to the display processor. Those problems introduced by this relationship are discussed further below.

The Multics/GDM interface procedures are capable of providing the following services:

1. Accept address request by segment number and offset of data to be displayed (GDM).
2. Convert this address to an absolute memory location for interpretation by the GIOC (GDM to Multics).
3. Transmit the datum from the GE-645 memory to the 338 (Multics).

In general, a GDM-type monitor requires only the simplest method possible of getting data from the time-shared computer to the display processor. On the Project MAC system, this means sending requests to a short I/O program running on the GE-645 GIOC. The 2400 bit per second Dataphone (201B modems) used for this transmission limits the request rate to approximately twenty per second (a negligible disturbance on a one-and-one-half microsecond per instruction processor). Higher data rate transmission can be used with corresponding increases in interference (if we increase the rate to 40,000 bps, the perturbance is still less than .1 percent) and special telephone lines.

All displays currently in use sample the GE-645 at rates at or near the available maximum. Displays with a number of data items occasionally resort to special

TABLE I—A display template to monitor a T-S user's activity

```
*address_table
tc_data                 /segment name
540; 541                /locations within the segment

*data_routines
a                       /name table of routines to be
                        /called by the GDM monitor
7777                    /end of table
a, 0                    /PDP-8 subroutine format
  call ge645, 1, 2      /get first data item
  call nplot, mp, 1     /plot "MP" state number
  call ge645, 1, 3      /get next data item
  call nplot, st ,1     /plot "ST" state number
  jms calc              /call to machine language
                        /subroutine to calculate bar
                        /graph length
  do hplot, bar         /plot horizontal bar graph
  call sked, 144, a     /reschedule "a" to be called
                        /by monitor in one second
  jmp i a               /PDP-8 subroutine format

*display-list
    dlstart             /macro instruction to start dis-
                        /play
    nl; nl              /"new line" for formatting
mp, 0                   /storage for "MP"
    sp2                 /spaces for display formatting
st, 0                   /storage for "ST"
    sp2                 /formatting
    hbar bar            /macro to create bar graph display
    escape              /display instruction macro
    top                 /display instruction macro to
                        /cause refreshing of display
```

sampling methods in order to update important items at least once a second: about the rate at which the human eye can follow a dynamic display with that much information.

*Advantages and disadvantages of GDM*

Advantages, disadvantages, capabilities, and limitations of GDM can be grouped into two categories: those relating to its monitoring ability; and those relating to its ability to report the information monitored.

## Monitoring ability

Several factors determine the usefulness of any type of monitor. These include the number and type of events it can monitor, the rate at which it can monitor them, and the interference that this observation will cause to the system being monitored.

One of the capabilities of GDM is a facility to change the point of observation easily: this is accomplished through the use of the display template. A new display template can be designed and operational in a short time and, once constructed, can be added to a library for future recall. No hardware changes need be made, no plug boards rewired, no probes changed to monitor a new or different event. Another display template with a few basic instructions is all that is needed to change the "probe" of GDM.

GDM, as constructed, is a sampling monitor. Current dataphone connections limit requests for data items to about twenty per second as mentioned above. Faster dataphone, direct connections or other means can be used to influence sampling rate. The current rate is such that "microsecond" events cannot be monitored. Transient data items will be missed if their core location changes many times in a second. Current displays, therefore, limit themselves to observing only "wired" data, this is, data whose core location need be determined only once during a particular monitoring period although the data itself may change many times. As approximately 80 percent of the Multics supervisors, data bases fall into this category at the current time, this is not particularly restrictive.

Monitoring which requires the collection of a large number of statistics over a very short time period similarly is hindered by the current configuration although "long-time" statistics are collected and displayed by a number of display templates.

Under Multics, short-time event monitoring is performed by special software embedded in the Multics supervisor.[10] A GDM display is used to observe, in real time, the data base of this monitor in order to see the time build up of the statistics and to note any abnormalities that might be missed by observing averages after an hour or more of operation. In this way, the advantages of a real-time display are combined with monitoring embedded in the time-sharing system (which causes significant interference when turned on) to provide a very useful tool.

The area of system interference has already been discussed but one item should be emphasized. In the Multics configuration, GDM need take only GIOC time—not CPU time. In computer systems where this is not possible, interference will still be negligible if the GDM monitor "steals" only enough information to make a useful display. Five hundred cycles per second is still only .1 percent on a two-microsecond cycle time computer and this is more than sufficient for even the most complex display.

### Reporting ability

Output of information is another area in which flexibility is crucial. Displays in Figures 3, 4, and 5 show only numbers, characters, and bar graphs. Displays have also been constructed with the types of graphs shown in Figure 6 and many others have been suggested for particular applications. It has been found that displaying the same information in different ways often presents an entirely different picture of what is going on. The only price to be paid for this flexibility is programmer time and even then it is no more difficult to display a bar graph (or any other type) than it is to simply show a number. Several display templates showing the same data in different formats can be made almost as easily as a single one and the best added to the GDM library.

For those who desire hard copy, GDM, in its current configuration, offers only photographs of its displays (stopped at any instant of time, saved on tape for future reference or photographing). Plotters of various kinds could perhaps be connected in tandem with a dynamic display and requested to plot a particular instance, even while the CRT display is still changing. Here again, the designer is limited only by the hardware available and his imagination.

## CONCLUSIONS AND OBSERVATIONS

The GDM System at Project MAC has served in two major capacities:

1. As a monitoring "control center".
2. As a debugging tool.

The very nature of a multiple-access computer system makes it very difficult to determine at one location exactly what is happening at all terminals. The GDM display, conveniently located near the main body of Multics programmers, is readily consulted to determine the state of a rampant user program, the availability of secondary storage space, or just the general health of the system (a slave display might possibly be installed near the computer itself or in the office of the system administrator as well). Many system programmers have, at one time or another, brought up the GDM System on their own initiative to find out various, otherwise unobtainable, pieces of information (a "cookbook" instruction sheet has been provided for just this purpose). A visit to the GDM display is always included as part of the standard system tour for visitors.

As a debugging aid, GDM has been invaluable. It is responsible for the detection of many system bugs—often transient or time dependent—that were not easily isolatable by previously available means.

One of the features of GDM that has made it so useful is its ability to simplify the act of dynamic display creation to the point where this is no more difficult than writing a simple assembly language program. This flexibility has paid many times over for the effort of implementation.

Finally, GDM can be readily adapted for use with other time-sharing systems: only two Multics-dependent modules exist in the monitor and display templates can be designed to suit any system.

GDM was designed as an experimental system and as such has been very useful at Project MAC. Its use during a period of intense debugging of the Multics system has proven its development worthwhile.

## ACKNOWLEDGMENTS

The author would like to express his gratitude to Professor F. J. Corbató, advisor for his Master's Thesis, for his continued support and aid during the period of this work. Thanks are also due to many members of the Multics development group at Project MAC without whose help this work could not have undertaken and in particular, Thomas Skinner Noel Morris, and Professor J. H. Saltzer.

## REFERENCES

1 E G COFFMAN L C VARIAN
*Further experimental data on the behavior of programs in a paging environment*
CACM Vol 11 No 7 1968 471-474

2 F J CORBATÓ V A VYSSOTSKY
*Introduction and overview of the Multics system*
Proc FJCC 1965 185-196

3 R C DALEY J B DENNIS
*Virtual memory, processes, and sharing in Multics*
CACM Vol 11 No 5 1968 306-333

4 D J EDWARDS
*GE-645 core memory X-ray program*
Multics System Programmers' Manual Section BE.13
Cambridge Mass MIT Project MAC internal doc 1966

5 G ESTRIN L KLEINROCK
*Measures, models, and measurements for time-shared computer utilities*
Proc ACM Nat Meeting 1967 85-95

6 G ESTRIN et al
*SNUPER COMPUTER, a computer in instrumentation automation*
Proc SJCC 1967 645-656

7 E L GLASER J F COULEUR G A OLIVER
*System design of a computer for time sharing applications*
Proc FJCC 1965 185-196

8 J M GROCHOW
*The graphic display as an aid in the monitoring of a time-shared computer system*
Project MAC Tech Rpt MAC-TR-54 Thesis Cambridge Mass Sept 1968

9 J F OSSANA L E MIKUS S D DUNTEN
*Communications and input/output switching in a multiplex computing system*
Proc FJCC 1965 231-240

19 J H SALTZER J W GINTELL
*The instrumentation of Multics*
Presented at the Second ACM Symposium on Operating System Principles Princeton N J 1969

11 A L SCHERR
*An analysis of time shared computer systems*
Project MAC Tech Rpt MAC-TR-18 Thesis Cambridge Mass June 1965

12 F D SCHULMAN
*Hardware measurement device for IBM System/360 time sharing evaluation*
Proc ACM Nat Meeting 1967 103-109

# A graph manipulator for on-line network picture processing

by HUGO A. DI GIULIO

*Stanford University*
Stanford, California

and

PAUL L. TUAN

*Stanford Research Institute*
Menlo Park, California

## INTRODUCTION

This paper describes research which involves the use of interactive computer graphics for processing systems analysis networks. The term "systems analysis network" is used to include project scheduling, task-resource simulation, computer programming flow diagrams, decision tree, assembly line balancing, flows in networks, etc. These network pictures usually characterize the precedence relations and the logical and data flow among network component parts, and are traditionally the planning tools for industrial engineers, operations research analysis, and management and systems planners. In this research, a system is developed to provide a "drawing board," through the use of interactive computer graphics, to compose, transform, decompose, partition, simplify, merge, and regenerate network pictures for the purpose of facilitating rapid convergence in man-computer experiments.

First, a study of the characteristics of network pictures, in the light of graph theory, is conducted. It provides a theoretical framework within which interactive graphics operations can be structured. Next, a system of representing and processing network pictures through boolean matrix operations is developed. This is followed by the development of algorithms with which to regenerate network pictures, such a picture would be isomorphic with its original drawing, while, at the same time, maximizing its visual effectiveness. Finally, a system which enables the user to perform various manipulation and transformation schemes is described.

This research is in connection with the Biotechnology Laboratory of the Department of Industrial Engineering at Stanford University. An ADAGE computer system (AGT/30) with an on-line graphics terminal is being used under the sponsorship of N.I.H. project NLM 00525-2 and School of Engineering, Stanford University.

### *Characteristics of network pictures for systems analysis*

In this study we shall limit our attention to only the following types of network:

Activity network (e.g., PERT, CPM)
Project scheduling
Job-resource simulation
Flows in networks (e.g., maximal flow, shortest route)
Decision tree
Computer program flow diagrams
Assembly-line balancing

For convenience, henceforth they will be grossly called "systems analysis networks," or "SA networks." The logical structure of these networks gives rise to some common characteristics in their graphic representation. We shall describe some of them below:

### Independence of geometric constraints

By independence of geometric constraints we mean that an SA network picture does not require rigid coordinate positions for its picture parts as is in the case of drawing of a physical object. An SA network is essentially a directed line graph[3] with only precedence relations to be considered. In fact, it can be constructed with only nodes and arcs. A node generally represents an event, a machine, an operation etc.; and an arc may represent an activity, a flow, and at the same time, gives a sense of precedence.

The order in which operations or decisions are performed in an SA network is expressed by precedence relations. A precedence relationship exists between nodes belonging to the same path. We say node x precedes node y to imply that y cannot occur until x has occurred. This relation may be expressed by precedence operators with symbols $>$, $<$. The expression $x > y$ implies x precedes y, or equivalently, $y < x$ (y is preceded by x). The precedence relationship is transitive, i.e., if $x > y$ and $y > z$, then $x > z$. All nodes in the same network which can relate to each other in this manner belong to a partial ordered field which we shall call a "transitivity closure." An immediate precedence relationship between two nodes is represented by $\rightarrow$, or $\leftarrow$. $x \rightarrow y$ implies x immediately precedes y; $x \leftarrow y$ implies x is immediately preceded by y. The $\rightarrow$, $\leftarrow$ operators (link operators) do not have transitive properties.

Therefore, an SA network picture can be defined the same as a directed graph which we shall denote by G. $G = (X, F)$ where F is a "precedence function" defined over X. X is the set of all nodes in G. F(x) is the set of all immediate successors of node x in G. The expression $y \in F(x)$, or simply $y \in Fx$, implies that node x and node y (both belong to X) are connected by a directed arc (an arrow) pointing from x toward y. We denote this arc by (x, y) where x is called the first node, and y the second node. x and y are called "adjacent nodes." (Henceforth, an individual node will be identified by a lower case English alphabet, with or without subscript).

The letter A denotes the set of all arcs in G. The expressions $G = (X, F)$ and $G = (X, A)$ are equivalent.

F is not necessarily a single valued function, for example, we may have $F(x) = \{u, v, w\}$, i.e., there are three arcs emanating from node x: (x, u), (x, v), and (x, w). $F^{-1}$ is an inverse function (the set of all immediate predecessors) where $F^{-1}(y) = \{x | y \in F(x)\}$. Thus, if (u, y) and (x, y) are the arcs with y as their second nodes, then $F^{-1}(y) = \{u, x\}$.

The functions $F^2, F^3, \ldots, F^n$ are defined by: $F^2x = F(Fx)$, $F^3x = F(F^2x)$, $\ldots$, $F^nx = F(F^{n-1}x)$. Likewise, $F^{-2}y = \{x|y \in F^2x\}$, $F^{-3}y = \{x|y \in F^3x\}$, $\ldots$, $F^{-n} = \{x|y \in F^nx\}$. $F^nx$ is called the "nth generation successor set of x," $F^{-n}x$, the "nth generation predecessor set of x."

To preserve the consistency of transitivity relationship (so that y cannot be both a successor and a predecessor to x) we shall regard all SA networks as being acyclic (i.e., there is no directed cycle in G). The cyclic conditions may be treated as acyclic with the use of "equivalent nodes" as will be discussed later.

We call $\hat{F}x$ the "successor set" of x. $\hat{F}x = \bigcup_{i=1}^{h} F^ix$ where $F^hx \neq \phi$ and $F^{h+1}x = \phi$. We call $\check{F}x$ the "predecessor set" of x. $\check{F}x = \bigcup_{i=1}^{k} F^{-i}$ where $F^{-k}x \neq \phi$ and $F^{-(k+1)}x = \phi$. We define the "forward transitivity closure" of x by $\{x\} \cup \hat{F}x$, the "inverse transitivity closure" of x by $\{x\} \cup \check{F}x$, and the "transitivity closure" of x, $\bar{F}(x)$, by $\{x\} \cup \hat{F}(x) \cup \check{F}(x)$.

### Subgraphs, partial graphs, partitions, and reduced graphs

A subgraph of $G = (X, F)$ is a graph $(Z, F_z)$ where $Z \subset X$, and for all nodes x in Z, $F_zx = (Fx) \cap Z$. i.e., a subgraph of G is the result of taking away at least one node, and its associated arcs, from G. A partial graph of G is a graph of the form $(X, F')$ where $F'x \subset Fx$ for all x in X, i.e., a partial graph of G has all the nodes of G but without some (at least one) of its arcs.

$X_1, X_2, \ldots, X_r$ constitute a partition of X if: (1) $\bigcup_{i=1}^{r} X_i = X$; (2) for every i and j, $i \neq j$, and $i, j \leq r$, $X_i \cap X_j = \phi$. A graph $G^\circ = (X^\circ, A^\circ)$, where $X^\circ = \{X_1, X_2, \ldots, X_r\}$, and $A^\circ$ is the set of arcs, is called a reduced graph of G. $(X_i, X_j) \in A^\circ$, $i \neq j$, if and only if there exist a node $x \in X_i$ and a node $y \in X_j$ such that $(x, y) \in A$.

### Common basic diagrams

Basic diagrams are subgraphs which possess certain topological characteristics into which an SA graph can be decomposed. We consider all SA graphs as aggregates of some basic diagrams. It is advantageous that these basic diagrams be prestored in a dictionary, therefore, it is not necessary to enumerate the topological details of a basic diagram each time it occurs in a graph. A graph may be collapsed into a simpler form by reducing the number of nodes and arcs in some, or all, of the basic diagrams contained in the graph (graph reduction is explained in the last section of this paper). We introduce some of the most commonly used basic diagrams below:

**Closed Serial Path** (see Fig. 1.a): Many SA networks are constructed with individual paths (e.g., job-resource simulation). Serial path is a simple and elementary path R(x, y) having n nodes and exactly n — 1 arcs. x and y are exterior points of R(x, y). Nodes in R(x, y) which are successors of x and predecessors of y are interior points of R(x, y). A closed serial path has the properties of: (1) Fx and $F^{-1}y$ are singletons; (2) For each interior points z, Fz and $F^{-1}z$ are singletons. A serial path which violates the aforementioned properties is an open serial path. The simplest form of a closed serial path is two nodes linked by an arc. In such a case, there is no interior point.

**Simple Out-Branch** (see Fig. 1.b): Branches in SA networks often indicate decision points. An out-branch occurs at a node x if $|Fx| > 1$. A simple out-branch requires that there is no path between members of Y where Y is a subset of F(x). In our example (Fig. 1.b), Y = {a, b, c}.

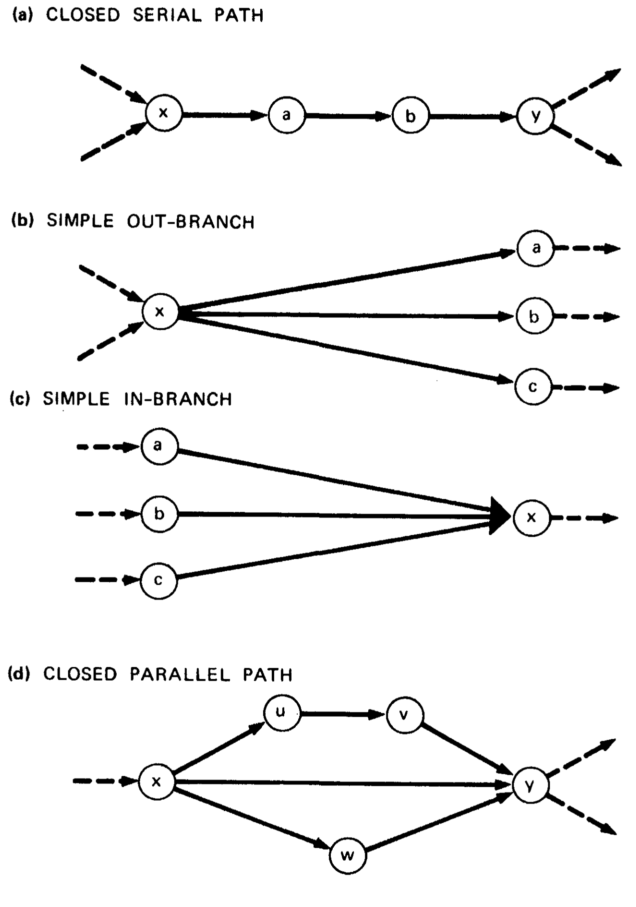


Figure 1—Basic diagrams

**Simple In-Branch** (see Fig. 1.c): In-branch occurs at a node x if $|F^{-1}| > 1$. To be a simple in-branch there must not exist a path between members of W where W is a subset of $F^{-1}(x)$. In our example, W = {a, b, c}.

**Closed Parallel Path** (see Fig. 1.d): A closed parallel path PP(x, y) implies that there are more than one closed serial path from x to y with x, y as their exterior points.

### The concept of weighted arcs and nodes

In an SA network an arc serves two functions: (1) To connect two nodes and give a sense of precedence, and (2) to carry values. For examples: cost, capacity, distance, flow units, data string, time, speed, probabilities, are values which may be associated with an arc. We consider those arcs which serve both functions weighted arcs. Similarly, a node may be weighted. For example, in a job-resource simulation a node is typically a processing station which contains channel capacity, mean processing rate, probability distribution function, queue storage, etc.

## *Picture composition and storage*

### Picture composition

To begin an interactive experiment a user must be able to draw a diagram on the CRT similar to the kind of diagrams he usually draws on paper. An SA network picture may be drawn by either manual input from the ADAGE graphics terminal, using joystick and light pen, or by programmed statements, or a combination of both. During the drawing phase of the experiment the console input involves the permanent display of certain "function keys" and "graphic primitives" on the bottom and the right edge of the CRT screen. A graphic primitive (one of the node symbols) is picked up and moved to the desired position on the screen with the movement of the tracking cross which is directed by the console joystick. Directed arcs are created by connecting nodes with the movement of the joystick. Alphanumerical labels for each node may be entered via the console typewriter. Figure 2.a shows the free-hand drawing of a project scheduling[3] network picture on the CRT.

Equivalently, a picture may be composed by pro-

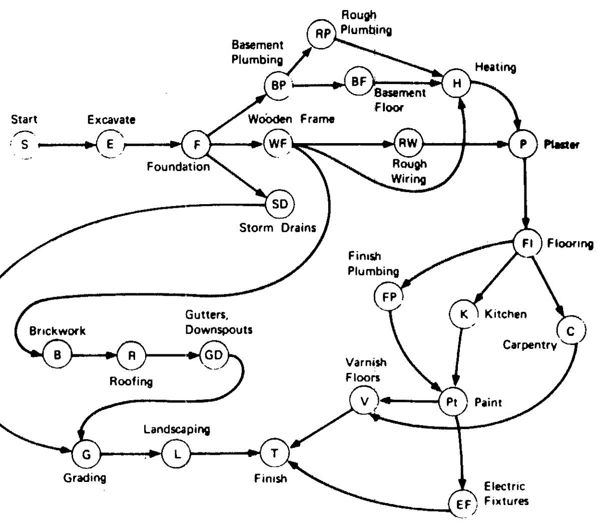


Figure 2a—Initial drawing of a project scheduling network

| | S | E | F | BP | WF | SD | RW | P | BF | H | FI | K | C | B | R | GD | G | L | T | V | Pt | FP | EF | RP |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S | | 1 | | | | | | | | | | | | | | | | | | | | | | |
| E | | | 1 | | | | | | | | | | | | | | | | | | | | | |
| F | | | | 1 | 1 | 1 | | | | | | | | | | | | | | | | | | |
| BP | | | | | | | | | 1 | | | | | | | | | | | | | | | 1 |
| WF | | | | | | | 1 | | | 1 | | | | 1 | | | | | | | | | | |
| SD | | | | | | | | | | | | | | | | | 1 | | | | | | | |
| RW | | | | | | | | 1 | | | | | | | | | | | | | | | | |
| P | | | | | | | | | | | 1 | | | | | | | | | | | | | |
| BF | | | | | | | | | | 1 | | | | | | | | | | | | | | |
| H | | | | | | | | 1 | | | | | | | | | | | | | | | | |
| FI | | | | | | | | | | | | 1 | 1 | | | | | | | | | 1 | | |
| K | | | | | | | | | | | | | | | | | | | | | 1 | | | |
| C | | | | | | | | | | | | | | | | | | | | 1 | | | | |
| B | | | | | | | | | | | | | | | 1 | | | | | | | | | |
| R | | | | | | | | | | | | | | | | 1 | | | | | | | | |
| GD | | | | | | | | | | | | | | | | | 1 | | | | | | | |
| G | | | | | | | | | | | | | | | | | | 1 | | | | | | |
| L | | | | | | | | | | | | | | | | | | | 1 | | | | | |
| T | | | | | | | | | | | | | | | | | | | | | | | | |
| V | | | | | | | | | | | | | | | | | | | 1 | | | | | |
| Pt | | | | | | | | | | | | | | | | | | | | 1 | | | 1 | |
| FP | | | | | | | | | | | | | | | | | | | | | 1 | | | |
| EF | | | | | | | | | | | | | | | | | | | 1 | | | | | |
| RP | | | | | | | | | | 1 | | | | | | | | | | | | | | |

Figure 2b—The connection matrix—C-matrix—of a project scheduling network

gramming. We introduce some of the commonly used operators, together with some examples below (with the contention that a network picture progresses from left to right):

**The Link Operators** (see Fig. 3 for examples):

| | |
|---|---|
| $x \rightarrow y$ | x links y and create arc (x, y). |
| $x \leftarrow y$ | x links from y and create arc (y, x). |
| $G_1(\{x_i\}) \rightarrow G_2(\{y_i\})$ | Graph $G_1$ links graph $G_2$. $\{x_i\}$ and $\{y_i\}$ are lists of concatenation points. For each $x_i$ there is a $y_i$ such $x_i$ links $y_i$. |
| $G_1(c) \rightarrow x$ | $G_1$ links x via (c, x) where $c \in G_1$, $x \notin G_1$. |
| $x \rightarrow [a, b, c]$ | Out-branch from x to a, b, c. |
| $[a, b, c] \rightarrow x$ | In-branch from a, b, c to x. |

**The Precedence Operators** (see Fig. 4 for examples):

| | |
|---|---|
| $x > y$ | x precedes y |
| $x < y$ | x is preceded by y |

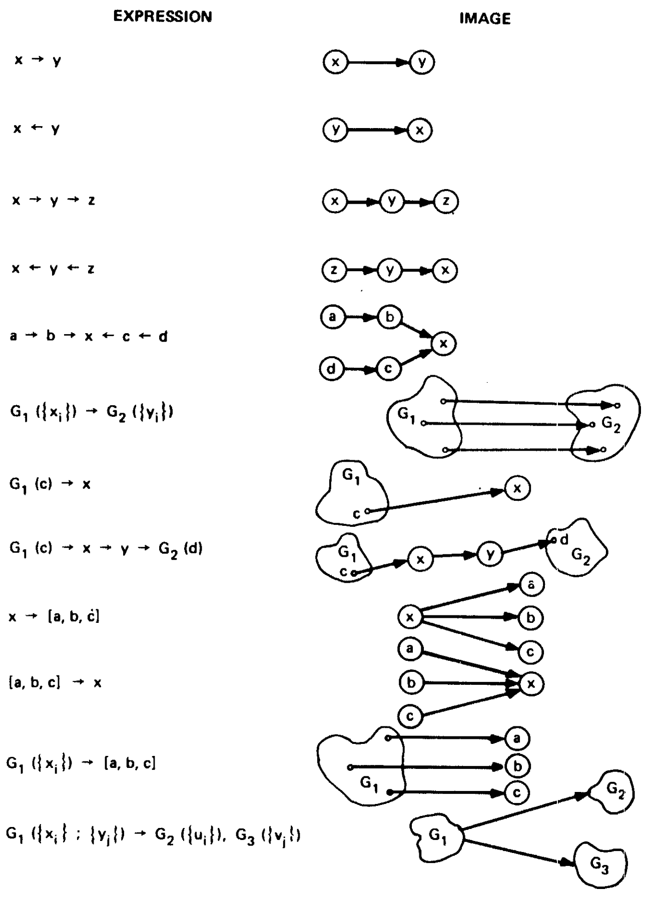


Figure 3—Examples of link operators

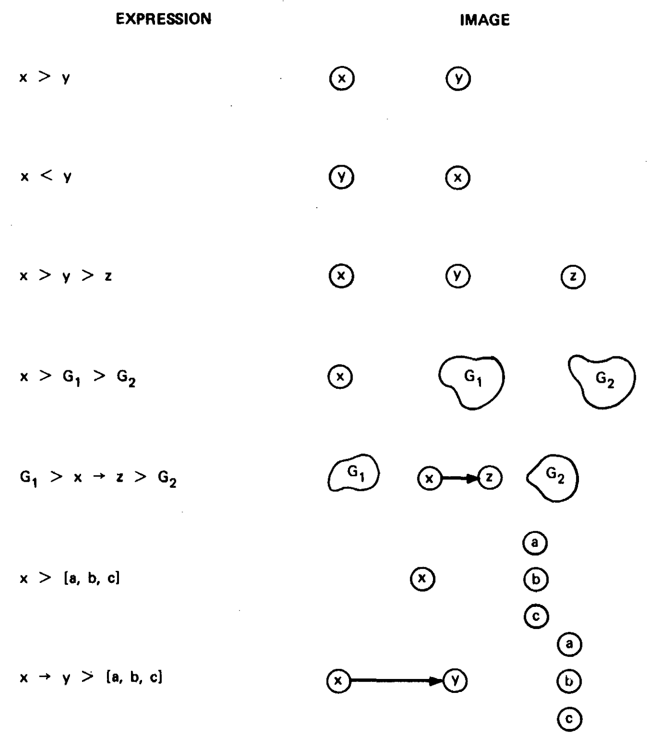


Figure 4—Examples of precedence operators

## Retention of picture information with Boolean matrix operations

The special topological characteristics of the SA networks (i.e., a picture is defined by the precedence and logical relationships among network components rather than their geometric attributes) permits us to make a radical departure from the conventional means of picture storage in which the coordinates and other geometric specifications (e.g., radius, angles) of vectors or primitives are to be remembered. The picture retention scheme for the SA networks involves a minimum amount of information, yet it preserves the isomorphism of the picture topology as well as the "meaning" of the picture. Under this scheme, a picture may be regenerated for the purpose of CRT display or for the purpose of revision, decomposition, reduction or merging with other pictures. This is done through the Network Picture Processing Language (NPPL) which employs boolean matrix operations for various picture manipulations.

During the picture composing phase, while a picture is being drawn on the CRT by the user, a "connection matrix," or "C-matrix" is constructed in the working storage of the computer. C-matrix is a boolean matrix[1] with dimension n x n where n is the total number of nodes in G. If we label the row corresponding to $x_i$ by i, and the column corresponding to $x_j$ by j, then the element of C has the value $c_{ij} = 1$ if $(x_i, x_j) \epsilon A$; $c_{ij} = 0$ if $(x_i, x_j) \notin A$. $x_i$ is a source node if $\sum_{k=1}^{n} c_{ki} = 0$ and $\sum_{k=1}^{n} c_{ik} > 0$. $x_i$ is a sink node if $\sum_{k=1}^{n} c_{ki} > 1$ and $\sum_{k=1}^{n} c_{ik} = 0$. $x_i$ is an isolated point if $\sum_{k=1}^{n} c_{ki} = \sum_{k=1}^{n} c_{ik} = 0$.

For example, Figure 2.b shows the C-matrix for the project scheduling network sketch given in Figure 2.a.

In order to preserve computer memory the C-matrix is converted into a "precedence matrix," or "P-matrix," before storage. A precedence matrix is a connection matrix with its row (columns) arranged in accordance with the precedence relations in G. The rules of arrangement are as follows:

1. If $y \epsilon \hat{F}x$, then y must be placed after x (i.e., the row (column) associated with y must have a larger index number than that of x).
2. If $y \epsilon \check{F}x$, then y must be placed before x.
3. If $y \notin \bar{F}x$, then the order between x and y is irrelevant.

Figure 5 shows the P-matrix of the project scheduling network of Figure 2.a. We notice that the P-matrix is triangular (this will always be true if the precedence relation are held), and it is predominantly inhabited with zeros. Both features contributed to the economical use of core storage.

*Picture regeneration*

### Convention of network picture arrangement

The network pictures stored in computer file may be retrieved in its entirety, or in part, for CRT display. It is also desirable to redisplay a picture immediately after it is drawn because invariably the computer will generate a "better" picture than the one drawn by the user. In our present effort the convention of a network picture generated by the computer includes the following rules:

1. All the source nodes are placed at the left end of the screen which means that the network pictures progress from left to right.
2. Only forward arrows are allowed, i.e., no backward or vertical arrows.
3. All arcs are made of linear segments.
4. Line crossings are to be minimized.
5. Other visual effectiveness considerations.

| | S | E | F | BP | WF | BF | RP | RW | H | P | FI | B | K | R | FP | SD | GD | Pt | C | G | L | V | EF | T |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S | | 1 | | | | | | | | | | | | | | | | | | | | | | |
| E | | | 1 | | | | | | | | | | | | | | | | | | | | | |
| F | | | | 1 | 1 | | | | | | | | | | | 1 | | | | | | | | |
| BP | | | | | | 1 | | | | | | | | | | | | | | | | | | |
| WF | | | | | | | | 1 | 1 | | | 1 | | | | | | | | | | | | |
| BF | | | | | | | | | 1 | | | | | | | | | | | | | | | |
| RP | | | | | | | | | 1 | | | | | | | | | | | | | | | |
| RW | | | | | | | | | | 1 | | | | | | | | | | | | | | |
| H | | | | | | | | | | 1 | | | | | | | | | | | | | | |
| P | | | | | | | | | | | 1 | | | | | | | | | | | | | |
| FI | | | | | | | | | | | | | 1 | | 1 | | | | 1 | | | | | |
| B | | | | | | | | | | | | | | 1 | | | | | | | | | | |
| K | | | | | | | | | | | | | | | | | | 1 | | | | | | |
| R | | | | | | | | | | | | | | | | | 1 | | | | | | | |
| FP | | | | | | | | | | | | | | | | | | 1 | | | | | | |
| SD | | | | | | | | | | | | | | | | | | | | 1 | | | | |
| GD | | | | | | | | | | | | | | | | | | | | 1 | | | | |
| Pt | | | | | | | | | | | | | | | | | | | | | | 1 | 1 | |
| C | | | | | | | | | | | | | | | | | | | | | | 1 | | |
| G | | | | | | | | | | | | | | | | | | | | | 1 | | | |
| L | | | | | | | | | | | | | | | | | | | | | | | | 1 |
| V | | | | | | | | | | | | | | | | | | | | | | | | 1 |
| EF | | | | | | | | | | | | | | | | | | | | | | | | 1 |
| T | | | | | | | | | | | | | | | | | | | | | | | | |

Figure 5—The precedence matrix—P-matrix—of a project scheduling network

As mentioned before, the special structure of the SA networks allows us to generate a network picture with an efficient generator which presupposes the topological characteristics of the graph, thus, reducing drastically the storage requirement. The graph generator of the NPPL operates on the P-matrix (or C-matrix) and transforms it into a graph image with all routes of interconnections "optimized." For example, Figure 6 is the same network picture as shown in Figure 2.a but it is interconnections "optimized." For example, Fig. 6 is the same network picture as shown in Fig. 2.a, but it is drawn with the convention and constraints set by NPPL.

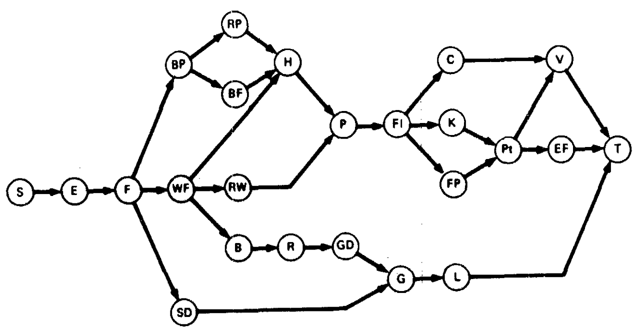


Figure 6—Regenerated picture of a project scheduling network

## Algorithms for optimum routing for interconnections

We consider the graph area as being a rectangle grid. The nodes of a graph are always placed at the intersection of the vertical and horizontal lines. We call the vertical lines "stages" and the horizontal lines "levels." If we can place each node of the graph at its proper stage and level, and connect them according to F function, then a graph is generated. Figure 7 shows the project scheduling network with each "stage" indicated. If the "stage" and "level" assignments are not properly made it may result into backward arrows and frequent occurrence of line crossings. Both features are undesirable from a visual effectiveness point of view. We shall briefly list the procedures of assigning stages as follows by using the example of the project scheduling network:

1. Place all source nodes in stage 0, S(0). e.g., S(0) = {s}.
2. Obtain $S'(1) = \bigcup_{x \in S(0)} F(x)$. e.g., S'(1) = {e}.
3. Obtain $S'(2) = \bigcup_{x \in S'(1)} F(x)$. e.g., S'(2) = {f}.

   In general, $S'(n) = \bigcup_{x \in S'(n-1)} F(x)$.
4. If $S'(n) \cap S'(n + k) = Y \neq \phi$ (where $k \geq 1$) then S'(n) would be modified by labeling members of Y in S'(n) as "dummy nodes," and the successors of dummy nodes would be deleted from any subsequent stages. The dummy nodes will be repeated at each succeeding stage until S'(n + k). The function of the dummy node is to be a "place marker" for an arc which crosses

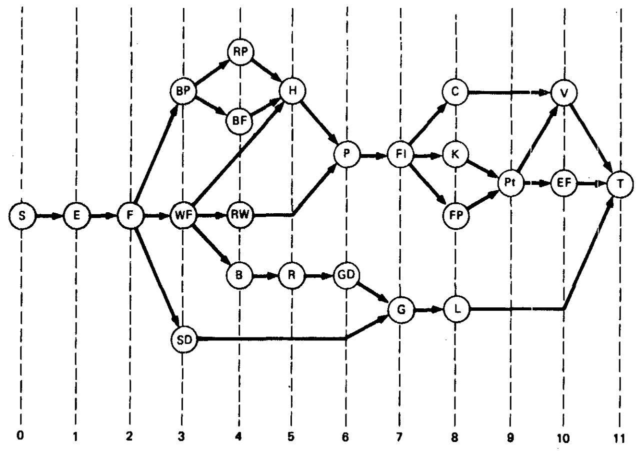


Figure 7—"Stages" of a network picture

S(0) = { S }
S(1) = { E }
S(2) = { F }
S(3) = { BP, WG, SD }
S(4) = { RP, BF, FW, H*, B, G*}
S(5) = { H, P*, R, G*}
S(6) = { P, GD, G*}
S(7) = { F1, G }
S(8) = { K, C, FP, L }
S(9) = { Pt, V*, T*}
S(10) = { V, EF, T* }
S(11) = { T }

*Dummy Node

Figure 8—Assigning stages

several stages (this is often necessary in order to avoid backward or vertical arrows) and to keep it free from interference from other arcs or nodes.

5. The "scan" process will continue until stage m is reached where S(m) = $\phi$, and all S'(j)'s, j = 1, 2, . . . , m — 2, have been modified (i.e., the labeling of dummy nodes and the deletion of their successors). The modified stages are then denoted by S(j) for all J($\neq$ 0). Figure 8 shows the result of the scan process as it applies to the project scheduling network.

The next is to assign "levels." Figure 9.b shows a graph of the project scheduling network from stage 7 through 11, using the order of node appearance in each stage (Figure 8) as the initial "level" assignment. Figure 9.a gives its associated P-matrix with stage partitions shown. As can be seen in Figure 9.b, that the "unoptimized" version of node positioning resulted into two line-crossings ((FP, Pt) with (C, V*), and (Pt, EF) with (V*, V)). The crossing violations can also be detected from the P-matrix as shown in Figure 9.b.

The P-matrix of Figure 10.a shows proper level assign-is shown in Figure 10.b.
ment (thus, the matrix is called P*-matrix). This is done by interchanging row (column) positions of K with C in stage 8, and Pt with V in stage 9. The optimized graph is shown in Fig. 10.b.

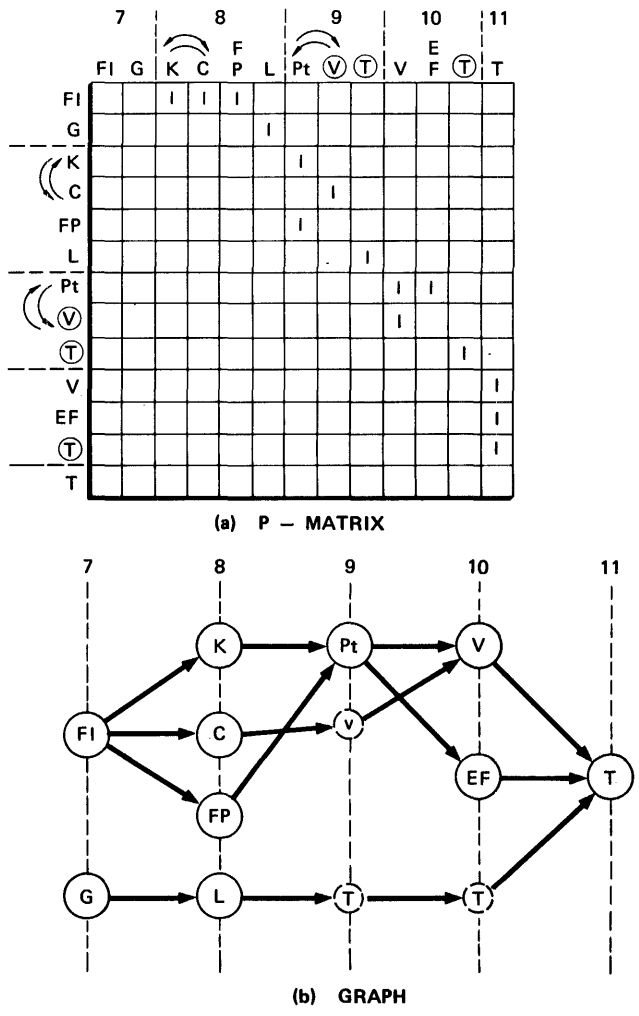


Figure 9—An example of improper "level" assignment

The criteria of optimizing the rows and columns of the P-matrix, in order to minimize line crossings, are:

1. Interchange rows and columns only within each stage.
2. The non-zero elements of each row should be consecutively located.
3. If the non-zero element of a row begins in column j, then no non-zero element of any previous row may begin in a column with column index less than j.

An optimal, or near optimal condition may be achieved by rearranging the columns and rows of P-matrix belonging to the same stage such that the resultant matrix meets, or most nearly meets, the above criteria.

The dummy nodes are not displayed on the CRT as full symbols, instead they are merely treated as point

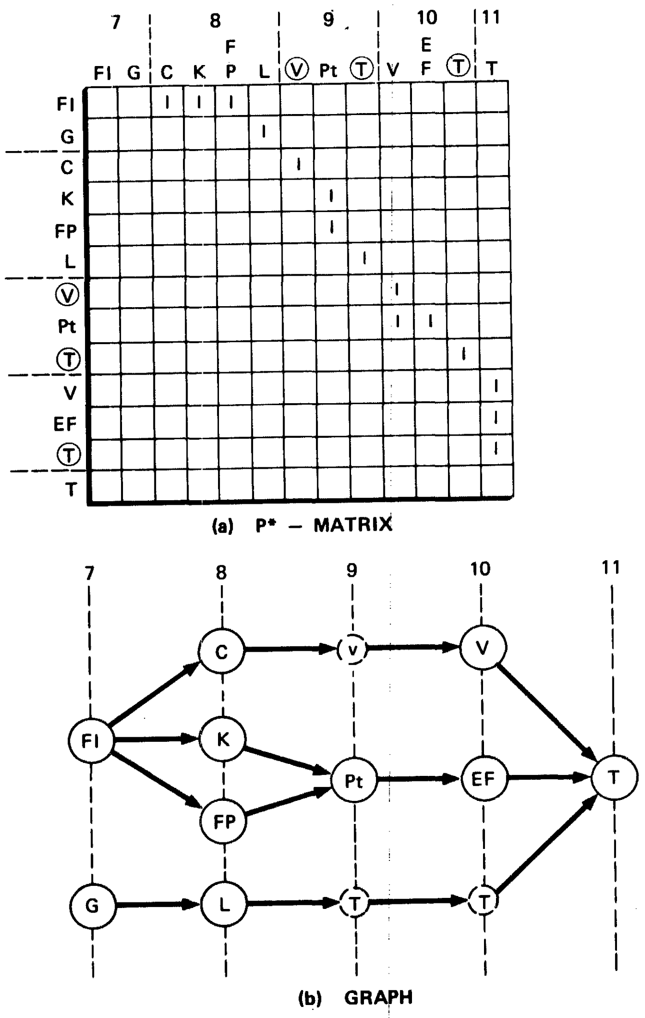


Figure 10—An example of proper "level" assignment

vectors which often serve as pivot points for arcs. For example, see arc (T*, T) in Figure 10.a.

## The handling of circuits

A graph which contains one or more circuits (i.e., directed cycles) is called a cyclic graph. A computer program flow diagram is typically a cyclic graph since program loops are the rule rather than the exception. Certain job-resource simulation models in which rejected product recycles back for rework at a previously encountered work station also constitutes a cyclic graph. While a cyclic condition can always be handled with connectors, thus making the resultant graph acyclic, it is more desirable to use directed arcs to display the actual circuits. In NPPL, the backward arrow, which represents the feed-back portion of the circuit, is treated as a forward arc during the graph regeneration phase. Upon completion of the line, an arrowhead facing the opposite direction is placed at the beginning of the arc (i.e., the left end of the arc). The presence or absence of circuits can be detected by examining the diagonal elements of the T-matrix. A non-zero diagonal element signifies the existence of a circuit.

*Picture manipulation*

## Union of graphs $G_1 + G_2$

Let $G_1 = (X_1, A_1)$ and $G_2 = (X_2, A_2)$. $G_1 + G_2$ is feasible if (1) $X_1 \cap X_2 = \phi$, or (2) $X_1 \cap X_2 \neq \phi$ and the set of common nodes (i.e., $X_1 \cap X_2$) have the same order of presence in both P-matrices $P(G_1)$ and $P(G_2)$. The procedure of obtaining the P-matrix associated with the graph $G_1 + G_2$, provided that $P(G_1)$ and $P(G_2)$ have been obtained, is as follows:

**If $X_1 \cap X_2 = \phi$:**

1. "Fill" Operation for $P(G_1)$—Expand $P(G_1)$ by adding $|A_2|$ zero row vectors of dimension $|A_1| + |A_2|$ to the bottom of $P(G_1)$, and $|A_2|$ zero column vectors to the right of $P(G_1)$, i.e., after the last column in $P(G_1)$. Thus, we have obtained the expanded $P(G_1)$, $\rho P(G_1)$.
2. "Fill" Operation for $P(G_2)$—Expand $P(G_2)$ by adding $|A_1|$ zero row vectors to the top, and $|A_1|$ zero column vectors to the left, of $P(G_2)$. Thus, we have obtained $\rho P(G_2)$.
3. Finally, $P(G_1 + G_2) = \rho P(G_1) \dot{+} \rho P(G_2)$. The symbol $\dot{+}$ stands for element-to-element boolean "inclusive-or" operations.

**If $X_1 \cap X_2 \neq \phi$:**

1. "Fill" Operation for $P(G_1)$—To obtain $\rho P(G_1)$ we fill $P(G_1)$ with zero row (column) vectors of dimension $|A_1| + |A_2|$ corresponding to those nodes which are in $G_2$ but are not in $G_1$.
2. "Fill" Operation for $P(G_2)$—To obtain $\rho P(G_2)$ we fill $P(G_2)$ with zero row (column) vectors of dimension $|A_1| + |A_2|$ corresponding to those nodes which are in $G_1$ but not in $G_2$.
3. $P(G_1 + G_2) = \rho P(G_1) \dot{+} \rho P(G_2)$.
   See Figure 11 for an example of $G_1 + G_2$.

## Intersection of graphs $G_1 * G_2$

Assuming $G_1 \cap G_2 \neq \phi$ we first obtain a compressed matrix $\gamma P(G_1)$ by striking out all rows and columns which are associated with non-common nodes. By

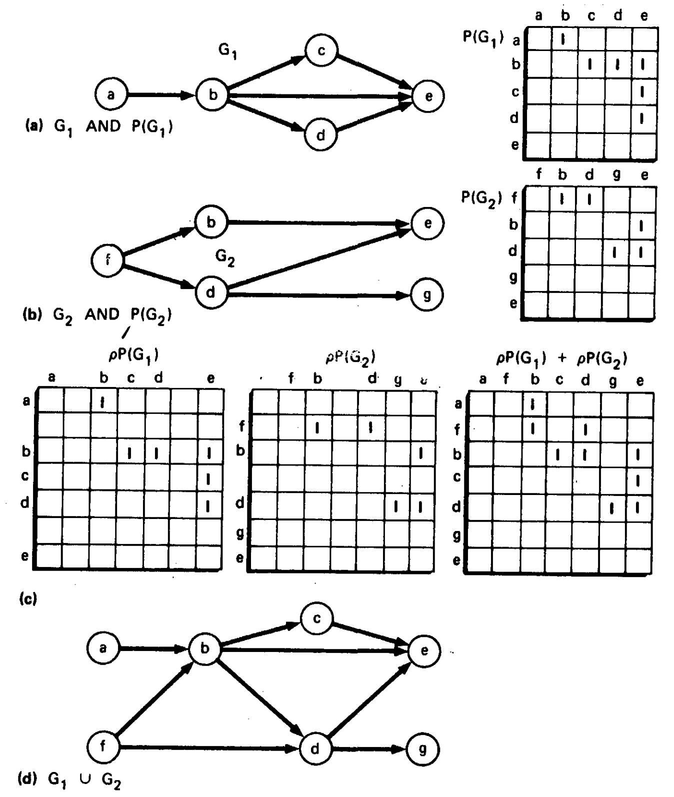


Figure 11—An example of $G_1 + G_2$

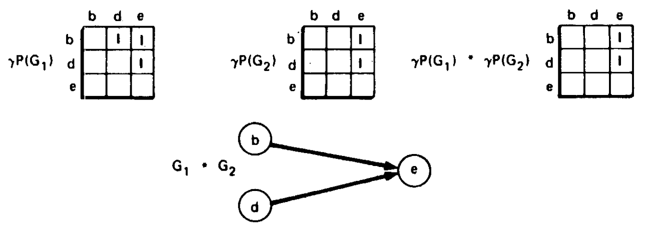


Figure 12—An example of $G_1 * G_2$

similar method, we also obtain $\gamma P(G_2)$. As in $G_1 + G_2$, nodes in $X_1 \cap X_2$ must have the same order in both P-matrices. $P(G_1 * G_2) = \gamma P(G_1) \mathbin{\dot{*}} \gamma P(G_2)$ where the $\dot{*}$ symbol indicates an element-by-element "and" operation of two boolean matrices, i.e., $A \mathbin{\dot{*}} B = C$ implies $c_{ij} = a_{ij} \wedge b_{ij}$. We use $G_1$ and $G_2$ as illustrated in Figures 11.a and 11.b to show the result of $G_1 * G_2$ in Figure 12.

## Deletion operations $G_1 - G_2$

Since $G_1 - G_2$ implies $G_1 - (G_1 * G_2)$ we strike out, from $P(G_1)$, all those rows and columns which belong to the set of common nodes, i.e., all the common nodes of $G_1$ and $G_2$, together with their associated arcs would be deleted from $G_1$.

The — operator is also an unary operator, e.g., the expression $-G_1$ produces the complement of subgraph $G_1$, or $G - G_1$, where $G_1 \subset G$; the expression $-A$ produces $G - A$ where $A \subset G$; $-x$ means the deletion of node x, together with its associated arcs, from G; $-(x, y)$ deletes the arc (x, y) from G.

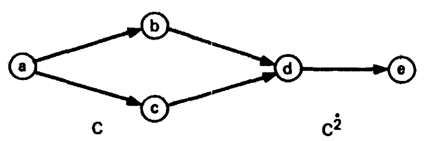
## Properties of +, *, and — operators

The operators described in the previous three sections have the following algebraic properties:

| | |
|---|---|
| 1. Commutative: | $G_1 + G_2 = G_2 + G_1$, $G_1 * G_2 = G_2 * G_1$ |
| 2. Associative: | $G_1 + G_2 + G_3 = G_1 + (G_2 + G_3) = (G_1 + G_3) + G_3$ |
| | $(G_1 * G_2 * G_3 = G_1 * G_2 * G_3) = (G_1 * G_2) * G_3$ |
| 3. Distributive: | $G_1 * (G_2 + G_3) = (G_1 * G_2) + (G_1 * G_3)$ |
| | $G_1 + (G_2 * G_3) = (G_1 + G_2) * (G_1 + G_3)$ |
| 4. De Morgan's Law: | $G_1 - (G_2 + G_3) = (G_1 - G_2) * (G_1 - G_3)$ |
| | $G_1 - (G_2 * G_3) = (G_1 - G_2) + (G_1 - G_3)$ |

## Generation of subgraphs

A standard operation in graph theory[1] is to compute $C^n$ which gives the number of paths of length n (length is defined as the number of arcs between two communicating nodes on a particular path) between any two nodes in G. As can be envisioned, this is a costly operation particularly if the graph is large. Instead of asking how many paths of length n between x and y we now ask whether there is a path from x to y of length n. We may achieve this by using boolean matrix operations. Thus, in raising C-matrix to a power we replace all ordinary summation by boolean summations. If P is the product matrix of A x B where matrices A and B have dimensions m x r and r x n respectively, and x is the symbol for boolean matrix multiply, then $p_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \ldots, + a_{ir}b_{rj}$ (where $p_{ij}$, $a_{ij}$, $b_{ij}$ are elements of matrix P, A, B respectively). We denote the nth power of C-matrix resulting from boolean matrix multiply by $C^{\dot{n}}$. $C^{\dot{n}}$ is a zero-one matrix. It is associated with a graph which possesses an arc (x, y) if and only if there exists a path of length n from x to y. For example, Figure 13 shows the C-matrix associated with a gaph and the powers of C. $C^{\dot{2}}$ shows that there exist paths from a to d, b to e, and c to e, of length 2. $C^{\dot{3}}$ shows that there is only one path of length 3 in the entire graph, and that



$$\begin{array}{c|ccccc} C & a & b & c & d & e \\ \hline a & & 1 & 1 & & \\ b & & & & 1 & \\ c & & & & 1 & \\ d & & & & & \\ e & & & & & \end{array} \qquad \begin{array}{c|ccccc} C^2 & a & b & c & d & e \\ \hline a & & & & 1 & \\ b & & & & & 1 \\ c & & & & & 1 \\ d & & & & & \\ e & & & & & \end{array} \qquad \begin{array}{c|ccccc} C^3 & a & b & c & d & e \\ \hline a & & & & & 1 \\ b & & & & & \\ c & & & & & \\ d & & & & & \\ e & & & & & \end{array}$$

Figure 13—Examples of $C^n$ matrices

$$\begin{array}{c|ccccc} & a & b & c & d & e \\ \hline a & & 1 & 1 & 1 & 1 \\ b & & & & 1 & 1 \\ c & & & & 1 & 1 \\ d & & & & & 1 \\ e & & & & & \end{array}$$

Figure 14—A transitivity matrix

is from a to e. $C^4 = 0$ which implies that there is no path of length 4 or more in the graph.

Boolean summation may also apply to the addition of connection matrices. $A \dotplus B = C$ (where A and B have the same dimensions) implies $c_{ij} = 0$ if $a_{ij} = b_{ij} = 0$; $c_{ij} = 1$ otherwise. Using the examples given in Figure 13 we show the result of $C \dotplus C^2 \dotplus C^3$ in Figure 14. The type of boolean matrix in Figure 14 which we shall call "transitivity matrix," or "T-matrix," indicates whether there is a path between any pair of nodes. It also gives $\hat{F}$ x and $\check{F}$x for any x. For example, the column labels associated with the non-zero elements of row vector b constitute the set Fb (i.e., {d, e}), and the row labels associated with the non-zero elements of column vector b constitute the set $\check{F}$b (i.e., {a}).

With the utility of the T-matrix there are a number of standard functions under NPPL for generating subgraphs. We shall introduce a few below:

| | |
|---|---|
| FTRAN(G, x) | Construct subgraph (X′, A′) of G = (X, A) where $X' = \{x\} \cup \hat{F}x$ |
| ITRAN(G, x) | Construct subgraph (X′, A′) where $X' = \{x\} \cup \check{F}x$ |
| TRAN(G, x) | Construct subgraph (X′, A′) where $X' = \bar{F}x$ |
| FTRAN(G, x, k) | Construct subgraph (X′, A′) where $X' = \{x\} \cup (\bigcup_{j=1}^{k} F^j x)$ |
| ITRAN(G, x, k) | Construct subgraph (X′, A′) where $X' = \{x\} \cup (\bigcup_{j=1}^{k} F^{-j} x)$ |

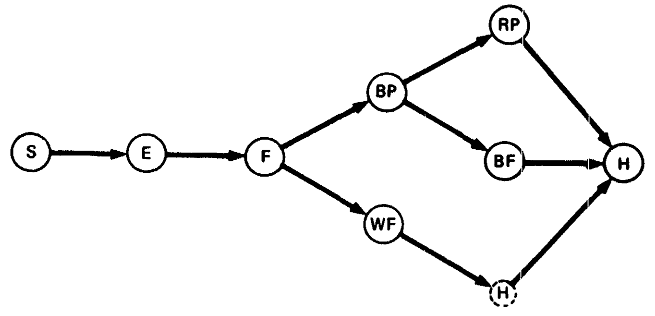
For example, if we name the graph in Figure 6 (a project scheduling network) $G_0$, the statement $G_1$ = ITRAN ($G_0$, H) would produce a graph showing all the activities which are prerequisite to the installation of heating (H), including the node H. Figure 15). The graph which represents activities between foundation (F) and flooring (F1) may be obtained by the statement $G_5$ = FTRAN ($G_0$, F) * ITRAN($G_0$, F1). (Figure 16)

## Graph reduction and expansion

Operating on the "basic diagrams" as explained earlier, the NPPL can successively reduce a graph to various levels of complexity as may be specified by the user. Some of the standard reduction functions are as follows:

RSPI(G, x, y) Reduce the interior nodes of

ALL ACTIVITIES PREREQUISITE TO HEATING
$G_1$ = ITRAN ($G_0$, H)



$$\begin{array}{c|ccccccccc} & S & E & F & BP & WF & RP & BF & (H) & H \\ \hline S & & 1 & & & & & & & \\ E & & & 1 & & & & & & \\ F & & & & 1 & 1 & & & & \\ BP & & & & & & 1 & 1 & & \\ WF & & & & & & & & 1 & \\ RP & & & & & & & & & 1 \\ BF & & & & & & & & & 1 \\ (H) & & & & & & & & & 1 \\ H & & & & & & & & & \end{array}$$

Figure 15—An example of ITRAN(G,x)

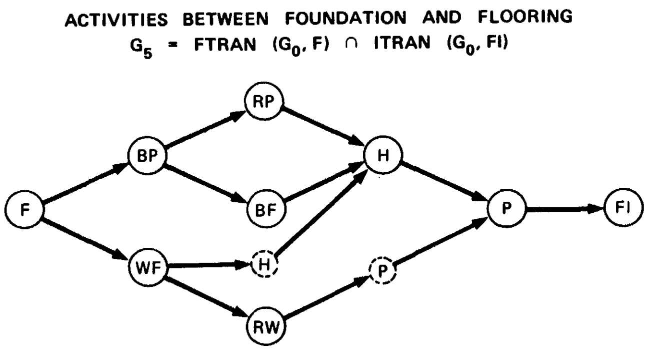

ACTIVITIES BETWEEN FOUNDATION AND FLOORING
$G_5$ = FTRAN $(G_0, F) \cap$ ITRAN $(G_0, FI)$
RP
BP
H
F
BF
P
FI
WF
H
P
RW

Figure 16—Activities between foundation and flooring

| | |
|---|---|
| | closed serial path R(x, y) in G and the arcs between them, into a single node. |
| RSP(G, x, y) | Reduce closed serial path R(x, y) into a single node. |
| ROB(G, x) | Reduce out-branches of node x in G such the set Y (defined earlier) is a single node. |

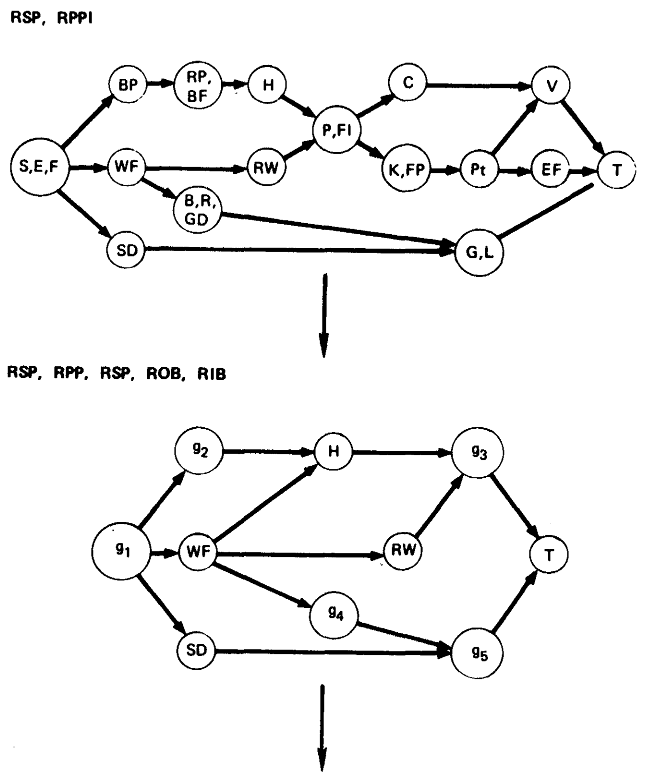


Figure 17—Examples of graph reduction

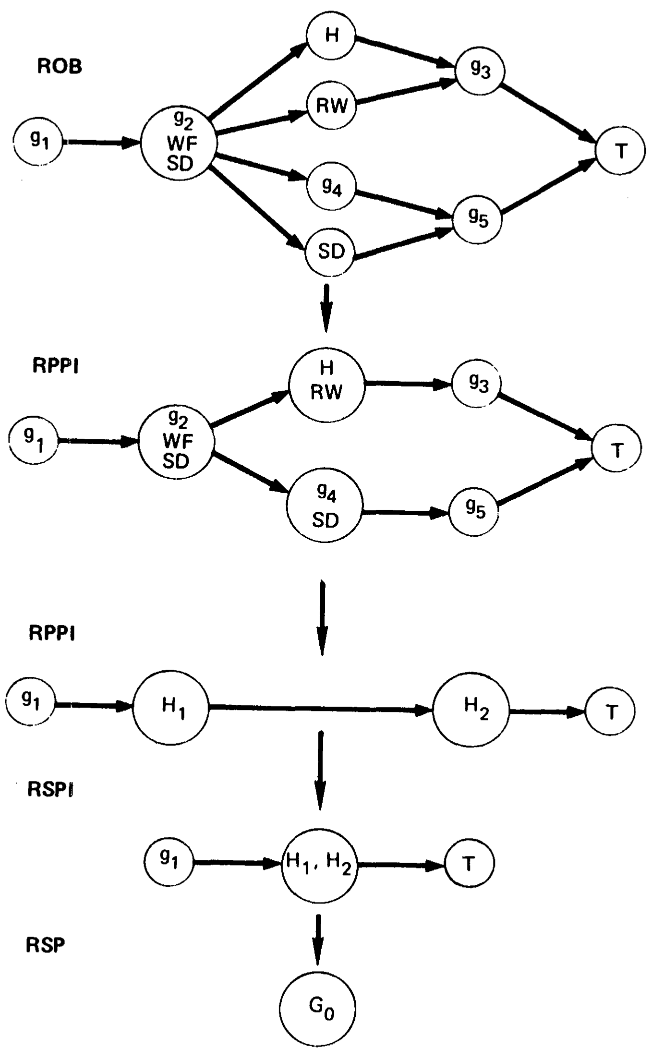


Figure 17—Examples of graph reduction con't.

| | |
|---|---|
| RIB(G, x) | Reduce in-branches of node x such the set W (defined earlier) is a single node. |
| RPPI(G, x, y) | Reduce interior nodes of closed parallel path PP(x, y) in G into a single node. |
| RPP(G, x, y) | Reduce closed parallel path PP(x, y) in G into a single node. |

As an example of utilizing some of the functions mentioned, Figure 17 shows a series of reduction beginning with the full graph of the project scheduling network G] of Figure 6.

Graph expansion is essentially the reverse of graph reduction. The nodes to be operated on must be compressed (macro) nodes. The expansion may be done in a single phase, or in several phases, e.g., the out-branches are expanded first, then the in-branches, then the serial paths, etc.

In conclusion, we would like to mention that the network picture processing language (NPPL) is not designed solely for the purpose of generating and manipulating network pictures. A greater objective is to provide an over-all control system through which man-computer experiments can be performed. We envision that once a picture is constructed (whether by initial composition, or by merging/decomposition operations) the user may assign (or reassign) input data to any node (or arc) by selecting the desired node (or arc) on the CRT. An input page would then appear on the CRT with pre-designated format to guide the user for inputing data. The matrix representation of each graph, as previously explained, would also serve as pointers to the storage areas of data pages. The simulation phase would then follow the input phase. During any phase of the experiment, controls may be returned to the picture composition and processing phase in order to maximize man-computer interaction.

## REFERENCES

1 C BERGE
*The theory of graphs and its applications*
John Wiley and Sons N Y 1962
2 E S BUFFA
*Modern production management*
John Wiley and Sons N Y 1965 538
3 R G BUSACKER T L SAATY
*Finite graphs and networks: An introduction with applications*
McGraw-Hill Book Co N Y 1965
4 H A DI GIULIO P L TUAN
*A system for network picture processing with interactive computer graphics*
Proc ACM 1969 Nat Conf and Exposition Aug 1969
5 C FLAMENT
*Application of graph theory to group structure*
Prentice-Hall 1963 Englewood N J
6 L R FORD D R FULKERSON
*Flows in networks*
Princeton Univ Press N J 1962
7 T R HOFFMANN
*Assembly line balancing with a precedence matrix*
Management Science July 1963 551-562
8 A KAUFMANN
*Graphs, dynamic programming and finite games*
Academic Press 1967 N Y
9 S C PARIKH W S JEWELL
*Decomposition of project networks*
Management Science Vol 11 No 3 1965 444-459
10 A C SHAW
*The formal description and parsing of pictures*
Stanford Linear Accelerator Center Rpt No 84 1968
Stanford Univ
11 A W STEINBERG
*Some notes on the similarity of three management science models and their analysis by connectivity matrix techniques*
Management Science Jan 1963 341-343
12 W M WAITE
*An efficient procedure for the generation of closed subsets*
CACM Vol 10 No 3 1967 169-171

# On-line recognition of hand-generated symbols*

*by* GEORGE M. MILLER

*University of California*
Berkeley, California

## INTRODUCTION

With the growth of information processing systems incorporating large data bases, many situations arise in which the data to be entered is a human's analysis of a problem. Often it will be undesirable to require the user to learn to type, and this mode can be cumbersome for random two-dimensional entry on a form or drawing. Using an electronic tablet coupled to a display tube would make it convenient for the user to point to a correct answer or print it in a very natural way. This paper describes a new technique for converting these hand written symbols to code words which can subsequently be processed by a computer.

It might be supposed that handwriting is not speed competitive with keyboard methods. Donald Devoe[1] of Sylvania's Applied Research Laboratory has recently conducted several experiments which indicate the contrary. Although handprinting of capitals and numerals is about five times slower than a skilled typist copying prose, the former compares favorably with the rate for untrained typists (i.e., about 60 characters/minute). In a task of making geometrical measurements on a drawing and recording this data in a table, Devoe found that handprinting required only about two-thirds of the time required using a keyboard. This difference was still evident with his subjects after six days of practice. Hence it may be anticipated for such application areas as computer-assisted instruction,[2] input of mathematical, logical and chemical formula in canonical forms, input and manipulation of matrices, program debugging,[3] specifying and designing systems by means of flowchart symbols, and two-dimensional game playing, that handprinting will not only be desirable to users, but also an efficient means of computer entry.

Research in hand-printed symbol recognition has been evident in the technical journals for more than a decade. The reader will soon discover that most symbol recognition literature is concerned with hard copy or off-line input. Typically, an optical scanner is used to obtain a two-dimensional array of points from a completed hand-printed character. The major effort of many researchers has been the exploration of unique methods of preprocessing or feature extraction to reduce the dimensionality of this raw data.[4] Others have placed greater relative emphasis on classification techniques and on the selection of features from a feature set or pool.[5] More recently several workers, including Duda and Hart,[6] have made use of context to improve recognition performance.

The electronic tablets used to obtain on-line source data provide a nearly exact trace of the path of the writing instrument and the order of the composite strokes used to inscribe a symbol. This time-sequence information is a great boon to machine recognition, but cannot be obtained by scanning off-line source images. For example, many individuals make 5's which look almost identical to their S's. However, an on-line recognizer will have no difficulty in distinguishing between this pair if the former is made with two strokes while the latter with only one. Similarly a

* This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract No SD-185.

lower case b and a numeral 6 are readily distinguished if their loops are inscribed in opposite directions of rotation.

Although on-line recognition systems have the advantage of low noise input data with higher information content, a number of challenges face the designer. He may desire a recognizer program which is invariant to size and position of the input symbol, has automatic means for detecting when a symbol is completed, is relatively insensitive to minor perturbations from ideal symbol shapes, has sufficient resolution to accommodate the wide range of symbols used in languages and the professions, is easily trained to the writing style of an individual, and which requires a minimum amount of memory space and computation time.

The author's on-line recognizer has been implemented on hardware typical of that found in a modern computer graphics environment. The components are shown in Figure 1 and include a time-shared computer, a CRT display, and a Rand Tablet.[7] When the pen switch is closed by pressing the stylus on the tablet surface, the sequence of filtered pen track coordinates, along with control bits to indicate the end of strokes, are temporarily stored in a buffer. In order to permit immediate display of the "ink trace," this operation is performed in a PDP-5 peripheral processor. The PDP-5 and the display control use a common core memory. Communication with the Berkeley Time Sharing System[8] (TSS) is by means of a half duplex link with a capacity of approximately 50K bits per second. The TSS schedules the user's dictionary building or recognition routines and has access to the PDP-5 memory. The user interacts with and controls these routines using either a teletype or by pointing to light buttons on the display. The recognition routine operates on the track coordinate data to determine when an input symbol has been completed and whether the symbol closely matches any of those previously stored in the user's dictionary. The output consists of an identification code, and data on the size and location of the recognized symbol.

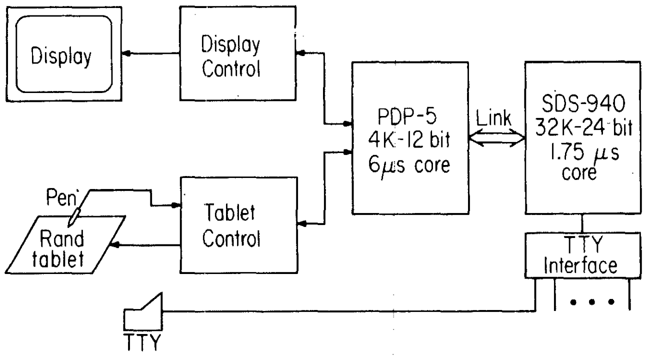


Figure 1—Hardware used in recognition research

## BACKGROUND

Two recognition algorithms developed by other researchers will be partially described in the next few paragraphs. (Additional background in on-line recognition techniques is contained in the dissertation from which this paper is abstracted[9]). The purpose of including this material is not to make a thorough comparison or evaluation, but simply to point out several limitations in their methods which led to a search for the techniques described in the body of this report.

G. F. Groner,[10] of the Rand Corporation, has developed an on-line recognizer which has successfully been applied to a larger system for creating, editing, and executing computer code and flow charts. Strokes are identified via a data-dependent sequence of tests determined by the system designer. The first four stylus directions are used to divide the strokes into groups. Further tests depend upon the particular subset of strokes and are chosen from the following: the number and/or relative position of corners, the relative position of starting and ending points, the number and/or positions of relative maxima and minima in y, and the fifth and succeeding stylus directions. The recognition of multiple-stroke symbols is based on correctly classifying the constituent strokes and their spatial relationships.

The Rand recognizer has several limitations. It cannot conveniently be modified for individual printing styles. Adding or deleting symbols is complicated because these operations frequently require changes in the tests used on resident symbols. The selection of features and the ordering of tests are based on an intuitive analysis of data obtained from a subset of potential users. There does not appear to be any convenient way of optimizing this design procedure.

M. I. Bernstein and T. G. Williams,[11] of the System Development Corporation, have recently described an on-line recognizer in which each user of the system may build a dictionary of the symbols he desires for his particular application. Strokes are divided into segments if they contain corners. Segments with a large or small aspect ratio are coded as vertical or horizontal lines respectively. Otherwise the segment is circumscribed with a minimum rectangle divided into the five sub-areas shown in Figure 2. The path of the segment is now retraced and each time a boundary is crossed, the number of the newly entered sub-area is added to a string to form an "area-sequence signature." In addition to the segment signature, the dictionary entries specify the geometrical relationship between the component segments and strokes of symbols. The distance between the center of each successor segment with respect to the center of the collection of its predecessors is quantized as coincident (C), proximate (P), or far (F). If the successor segment or stroke is proximate or far, the direction of its center with respect to the center of the collection of predecessors is quantized to one of eight sectors.

The SDC system requires an exact match of segment signatures and their spatial data for recognition so that a user's dictionary should contain all of the variations anticipated. As an example, Figure 2 shows that the first stroke of a numeral 4 could have three different area-sequence signatures. For each of these the second stroke could be in any of the four spatial positions shown. It is very unlikely that a particular user would produce all of the twelve possible combinations, but half this number is likely. Mr. Bernstein has indicated that on the average he requires three or four dictionary entries per symbol and that certain symbols require two or three times this number.

Although multiplicity of dictionary entries may not be a serious limitation of the SDC recognizer, it seemed desirable to this author to find a symbol descriptor and

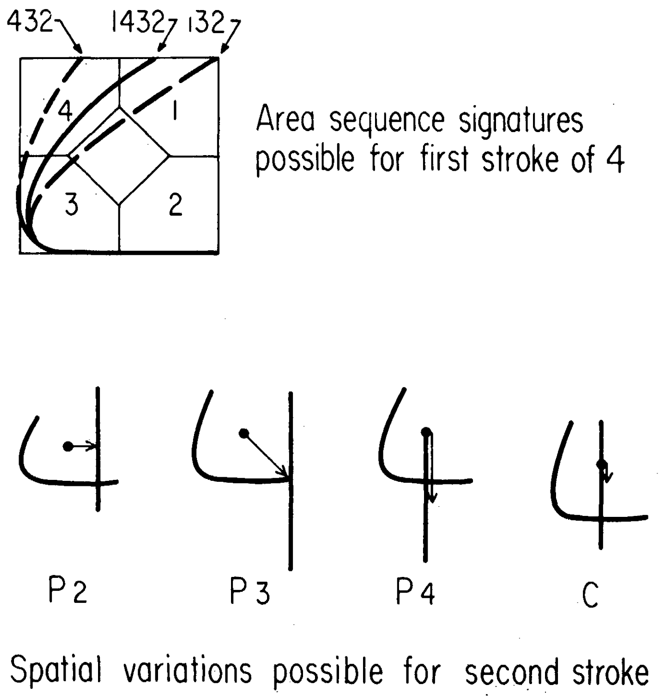

432
1432
132
4
1
3
2
Area sequence signatures possible for first stroke of 4
P2
P3
P4
C
Spatial variations possible for second stroke

Figure 2—Multiplicity of dictionary entries

recognition technique which would permit a high recognition rate but require fewer dictionary entries per symbol. Two concepts are used to obtain this goal. On the one hand it does not seem necessary or desirable to require a rigid geometrical relationship between the component strokes or segments of a symbol unless this information is needed for classification. If the numeral 4 is the only symbol which is generated using a two-stroke sequence similar to L followed by I, then there is no need to require any particular spatial relationship between the strokes. It follows, however, that some sort of automatic procedure is needed to determine which spatial information in a large set of symbols is redundant. A second way to reduce the number of dictionary entries is to devise a segment signature scheme which lends itself to the use of bestmatch techniques. The idea is to compute the degree of simliarity of an input segment with a set of prototypes and choose the closest match. With this capability it should not be necessary to store combinations of moderately distorted segments, but only nominal shapes.

*CVS signature and Lee metric*

### General description

In both of the above-mentioned recognition methods, an input symbol is classified on the basis of a number of discrete decisions. As a general principle it seems preferable to retain full information at each intermediate stage in the symbol recognition process.[12] Stated another way, it is desirable to have a smooth transformation between data spaces. A segment descriptor can be thought of as performing a mathematical transformation on the sequence of pen track coordinates. The idea of a smooth transformation is analogous to that of a continuous transformation in the mathematical sense.

The argument for the principle of smooth transformations can be made by an example. Consider as two segment classes the numeral I and the right angle L. As the lower half of an ideal I is rotated counterclockwise the generated symbol will pass through a transition region where the probability of its being in class L increases and the probability of its being in class I decreases. A good segment descriptor and classification method should reflect this continuous change. In the case of handwritten symbols, it is also desirable to have a feature space which is invariant to symbol size and position.

The author of this paper has conceived and tested a segment descriptor and metric which obtains the goals mentioned in the previous paragraph. This new method employs what will be called the contour vector sequence (CVS) and has some similarity to an encoding scheme described by Freeman.[13] In Freeman's method a square mesh is superimposed on the arbitrary curve to be encoded. Mesh nodes lying closest to the intersections between the curve and the mesh define a straight-line approximation to the given curve. The scheme is illustrated for two symbols in Figure 3. Successive nodes can only be one of eight, so the resulting encoding is a sequence of octal digits. The number of elements in the chain is directly proportional to the length of the curve. In a subsequent paper Feder and Freeman[14] use this encoding technique to fit a given curve to a similarly-shaped section of a larger curve. However the method is size variant and cannot be used for measuring the degree of similarity between two arbitrary segments.

In the author's CVS encoding scheme the contour of a segment is subdivided into six nearly equal length arcs which are approximated by their associated chords. Each of the chords is quantized to a vector having one of eight possible directions. Hence the resulting signature is a vector CVS = $s_1 s_2 s_3 s_4 s_5 s_6$ of six components, where each component takes slope of values between zero and seven. (See Figure 4.)

The degree of dissimilarity between two segments is obtained by summing the absolute rotational difference, expressed in angular units of $\pi/4$ radians, between corresponding components of the associated contour vector sequences. This distance measurement is equivalent to the Lee metric used in coding theory.[15] Specifically, if segment A has

$$\mathrm{CVS}_A = a_1 a_2 \cdots a_6 \tag{1}$$

and segment B has

$$\mathrm{CVS}_B = b_1 b_2 \cdots b_6 \tag{2}$$

then the Lee distance (will also be called the mismatch) between the segments A and B is given by

$$D_L(A, B) = MM_{A-B} = \sum_{i=1}^{6} |c_i| \tag{3}$$

where

$$c_i = |a_i - b_i| \tag{4}$$

$$|c_i| = \begin{cases} 8 - c_i, \text{ for } 5 \leq c_i \leq 7 \\ c_i, \text{ otherwise} \end{cases} \tag{5}$$

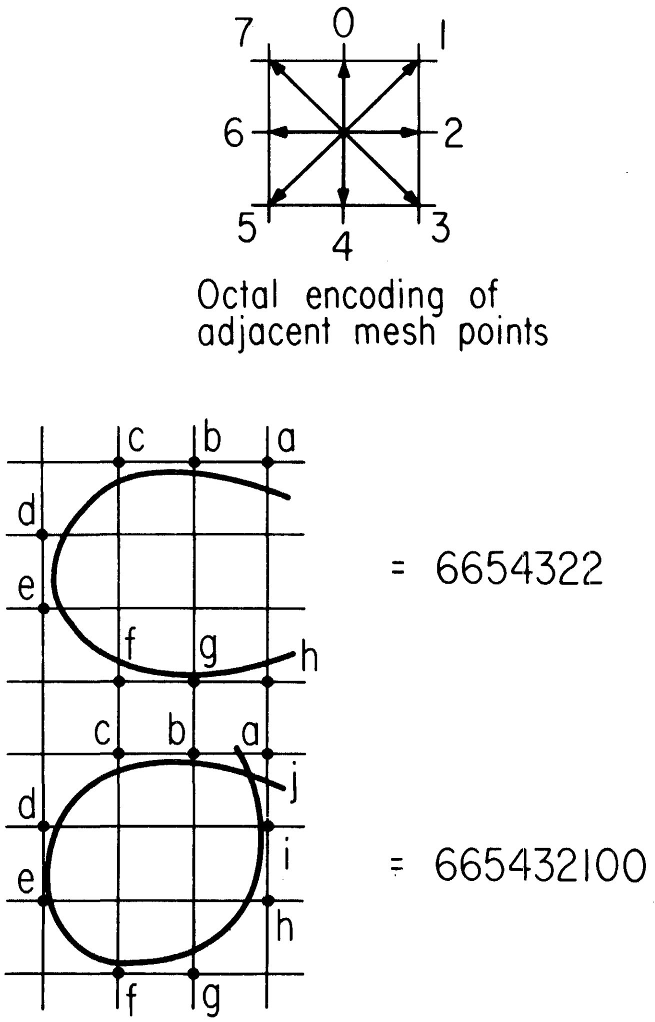


Figure 3—Freeman code

clearly

$$0 \leq |c_i| \leq 4 \tag{6}$$

As $|c_i|$ cannot exceed 4 angular units, the maximum Lee distance between two segments is 24.

Figure 4 illustrates the contour vector sequences for an alpha and a delta, and calculates a Lee distance of ten between these two symbols. The figure also shows a mismatch of only two between somewhat different alpha segments. The latter is an example of the smooth transformation between the pattern space and the CVS feature space. Data obtained from an experimental recognition program has shown that similar segments are mapped into points in the feature space which are close together in terms of the Lee metric. This clustering of segments which look alike to humans makes it

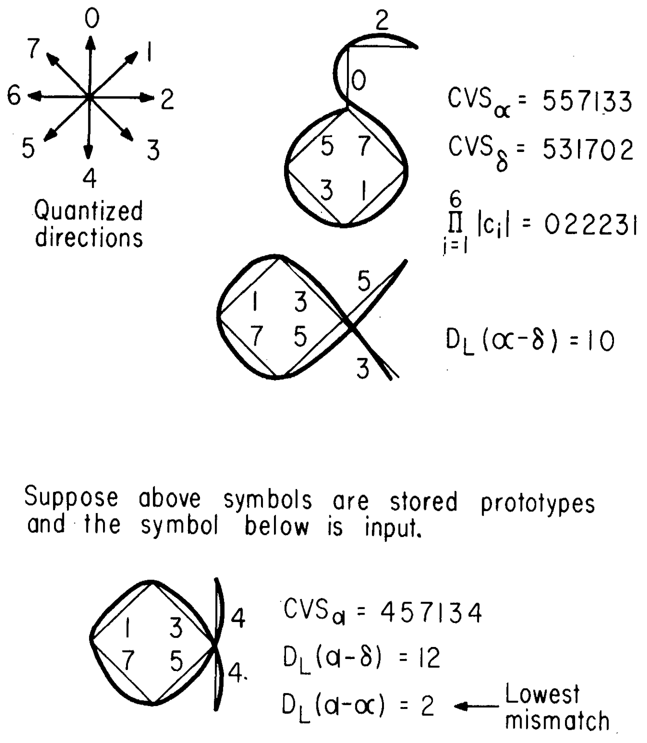


Figure 4—CVS encoding and mismatch calculation

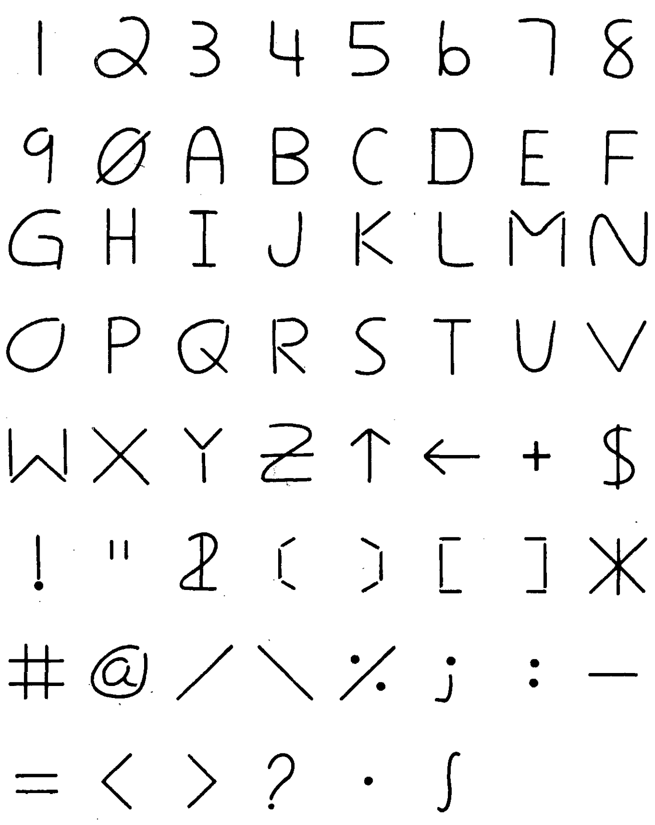


Figure 5—Hand printed teletype symbols

possible to only store nominal segments and use the metric to recognize non-ideal segments on a nearest prototype basis. The property also can be used to advise a person that certain symbol pairs which he creates are very "close" to each other and may give trouble in either human or machine recognition.

## Choice of six component CVS

Several factors contributed to the choice of six components in the CVS. In order to reduce storage requirements for a user's dictionary, it is desirable to use as few components as possible. On the other hand the CVS must provide sufficient resolution to distinguish between classes in a large set of symbols. Experiments were conducted with a variety of symbol shapes in order to obtain a compromise betwen these two goals.

As a minimal requirement it was felt that an on-line recognizer should accommodate handprinting of the teletype symbols shown in Figure 5. If strokes containing cusps (such as the 3 and 9) are subdivided into less intricate segments, this set of symbols can be conveniently printed using the 28 segments shown in Figure 6. As many of these prototype segments are symmetric about an axis, it appears desirable to have an even number of components in the CVS. Figure 6 gives visual evidence that very little shape information is lost if these segments are approximated with six components. This number of components also provides a minimum Lee distance four between any pair of segments. The symbol pairs having this lowest mismatch are 1 – ∫, C – <, S – ∫, and U – V.

Although a contour vector sequence having four components probably would be sufficient for many applications involving a small number of symbols, six components are needed to distinguish between the symbol pair S – ∫ of Figure 7. The figure also shows that only four components provide a rather poor straight line approximation of intricate curves such as the theta or the lower case e. A final factor affecting the choice of a six component CVS for further experimental investigations was the 24 bit word length of the computer, leaving six additional bits for other kinds of segment data.

## Computational algorithms

The computation of the contour vector sequence begins with a pre-processing operation on the raw pen data. The Rand Tablet (see Figure 1) has a resolution of .01 inches and is sampled each seven milliseconds

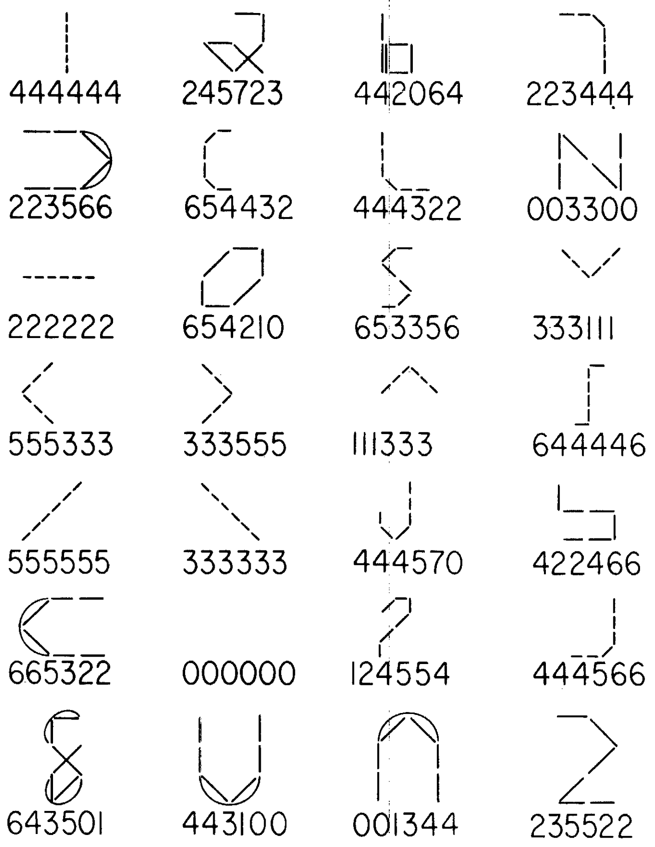


Figure 6—Segments used to print teletype symbols

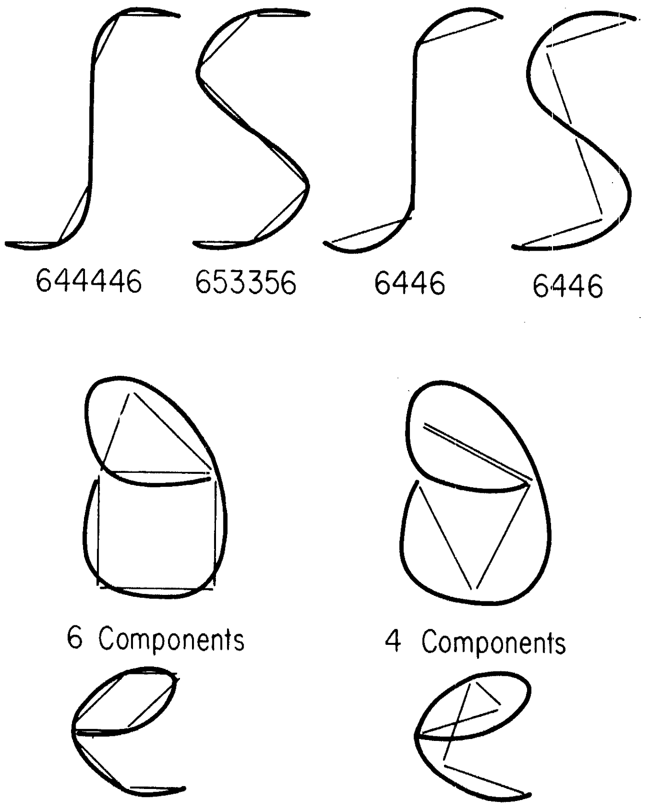


Figure 7—Four and six component CVS's

to obtain the location of the stylus. To reduce redundancy and filter out spurious noise from the tablet, the PDP-5 accepts a new coordinate $(x_j, y_j)$ only if the following three conditions are satisfied:

$$(x_j - x_i) < K_1 \quad (7)$$

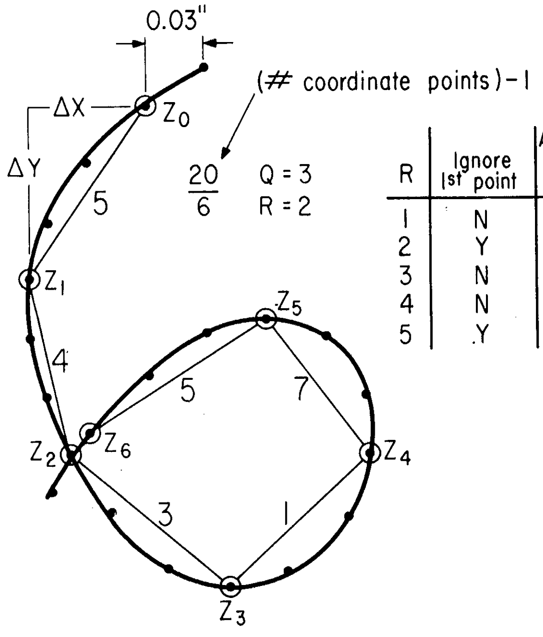
$$(y_j - y_i) < K_1 \quad (8)$$

$$[(x_j - x_i) \geq K_2] \text{ OR } [(y_j - y_i) \geq K_2] \quad (9)$$

where $(x_i, y_i)$ is the last coordinate accepted, $K_2$ defines an inner window, and $K_1$ defines an outer window. $K_1$ and $K_2$ are preset to ten and three respectively, but may be changed from the teletype using the command SET PARAMETERS. Owing to the high sampling rate, a new point is stored whenever the x or y coordinate changes .03 inches from the previously accepted point. This amount of resolution has been found sufficient for subsequent computations on 1/4 inch high symbols, but $K_2$ may be increased for larger symbols.

From the above mentioned filtering process the contour of a segment is represented by a sequence of nearly equally spaced x-y coordinates. These points are used to obtain a six-chord approximation to the segment. The algorithm (see Figure 8) consists of dividing the number of coordinate points less one by six, and taking the quotient (Q) as the nominal distance between adjacent chord points $(z_i, z_{i+1})$. If the division produces a remainder (R) it is distributed between the chords. If $R \geq 3$, an extra point is added when forming each of the odd chords $z_0z_1$, $z_2z_3$, and $z_4z_5$. If R is equal to one or four the last coordinate is removed and if R is equal to two or five the first is also neglected. These rules are summarized in a table of Figure 8.

The final step in computing the CVS signature is a quantization of the chord directions into one of eight sectors. The first component of the CVS is computed using the $\Delta Y$ and $\Delta X$ associated with the chord $z_0z_1$. In the example of Figure 8 $\tan 22.5° \leq |\Delta Y/\Delta X| \leq \tan 67.5°$, $\Delta X < 0$, and $\Delta Y < 0$, indicating the direction 5. A similar application of the Quantization Table to the remaining chords results in a signature of 543175 for the numeral 6 shown.

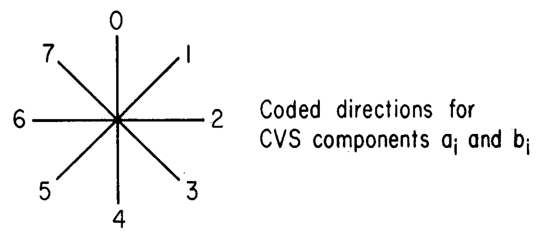


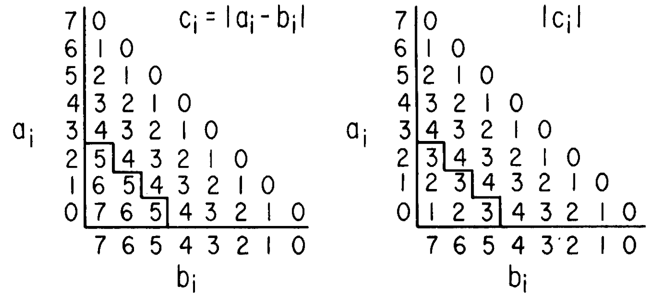
| R | Ignore 1st point | Add point to odd legs |
|---|---|---|
| 1 | N | N |
| 2 | Y | N |
| 3 | N | Y |
| 4 | N | Y |
| 5 | Y | Y |

| | ΔX>0 ΔY>0 | ΔX≤0 ΔY>0 | ΔX>0 ΔY≤0 | ΔX≤0 ΔY≤0 |
|---|---|---|---|---|
| tan 67.5° ≤ \|ΔY/ΔX\| ≤ ∞ | 0 | 0 | 4 | 4 |
| tan 22.5° ≤ \|ΔY/ΔX\| < tan 67.5° | 1 | 7 | 3 | 5 |
| 0 ≤ \|ΔY/ΔX\| < tan 22.5° | 2 | 6 | 2 | 2 |

Quantization table

Figure 8—Algorithm for computing CVS





$$c_i = |a_i - b_i| \qquad \begin{array}{c|cccccccc} a_i & & & & & & & & \\ 7 & 0 & & & & & & & \\ 6 & 1 & 0 & & & & & & \\ 5 & 2 & 1 & 0 & & & & & \\ 4 & 3 & 2 & 1 & 0 & & & & \\ 3 & 4 & 3 & 2 & 1 & 0 & & & \\ 2 & 5 & 4 & 3 & 2 & 1 & 0 & & \\ 1 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & \\ 0 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\ \hline & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\ & & & & b_i & & & & \end{array}$$

$$|c_i| \qquad \begin{array}{c|cccccccc} a_i & & & & & & & & \\ 7 & 0 & & & & & & & \\ 6 & 1 & 0 & & & & & & \\ 5 & 2 & 1 & 0 & & & & & \\ 4 & 3 & 2 & 1 & 0 & & & & \\ 3 & 4 & 3 & 2 & 1 & 0 & & & \\ 2 & 3 & 4 & 3 & 2 & 1 & 0 & & \\ 1 & 2 & 3 & 4 & 3 & 2 & 1 & 0 & \\ 0 & 1 & 2 & 3 & 4 & 3 & 2 & 1 & 0 \\ \hline & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\ & & & & b_i & & & & \end{array}$$

| $c_i$ | 5 ...0101 | 6 ...0110 | 7 ...0111 |
|---|---|---|---|
| 2's compliment of $c_i$ | $...1\underbrace{011}$ | $...1\underbrace{010}$ | $...1\underbrace{001}$ |
| $\|c_i\|$ | 3 | 2 | 1 |

Figure 9—Algorithm for computing $|c_i|$

If the number of coordinate points is less than seven, the CVS is computed in a different manner. Segments having four to six points are assumed to be straight lines and only the two end points are used. The quantized slope of the chord betwen these points is assigned to each CVS component. For example a short mid bar in the letter F would have a 222222 signature. If the number of coordinate points is in the range one to three, the segment is assumed to be a dot and assigned a CVS of 000000. Although this signature is also that of a vertical bar drawn bottom to top, the latter is not commonly inscribed. However, the dot could just as well be assigned any signature having a large Lee distance with respect to the other segments employed. By treating short segments in the manner described, it is possible to utilize a larger $K_2$ (inner window) and thereby reduce storage requirements and computational time.

The Lee distance between two segments is obtained by summing the absolute rotational difference $|c_i|$ between corresponding components ($a_i$, $b_i$) of the associated CVS's. As indicated by equation (5), $|c_i|$ cannot always be obtained by simply computing the absolute algebraic difference $c_i$ between $a_i$ and $b_i$. For example two components with quantized directions 7 and 2 have $c_i = 5$, but $|c_i| = 3$. Equation 5 also shows that when $c_i$ is 6 or 7, the respective $|c_i|$'s are 2 and 1. However the lower table in Figure 9 demonstrates that when $5 \leq c_i \leq 7$, the correct $|c_i|$ is obtained from the least three significant bits of the 2's complement of $c_i$. This simple algorithm has been implemented with standard machine instructions.

## Dissecting strokes into segments

It is well known in mathematics that continuous tranformations depend upon well behaved functions. If a function is not continuous and/or analytic, it may be necessary to apply the transformation separately to a piecewise approximation of the function. In an analogous way the smooth transformation property of the CVS signature and Lee metric is related to the geometrical properties of the two dimensional entities on which it operates. Discontinuities in the pattern space are easily handled because in an on-line imple-

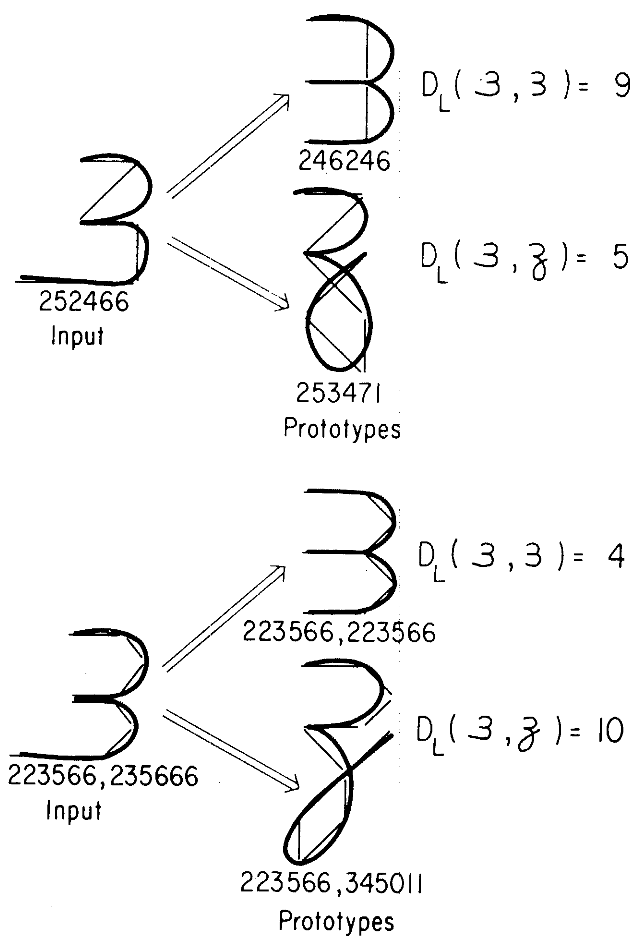


Figure 10—Improvement obtained by segmenting

mentation the start and end of a stroke are reliably indicated by a micro-switch in the writing stylus. Hence it is possible to compute a separate CVS for each component stroke in a symbol. The manner in which the Lee metric is applied to multi-stroke symbols is discussed in the next section.

Sharp corners or cusps in a stroke correspond to points in which the derivative of a function is not defined, and can be troublesome to the CVS transformation. A lower case z and a numeral 3 will be used as an example. Figure 10 shows that a slightly distorted 3 may have less mismatch with the prototype z than with the prototype 3. However, the lower half of the figure demonstrates that the smooth transformation property can be restored if each of the symbols are separated at the cusp into two segments. Now the upper left-cup common to both symbols has the same prototype CVS, and classification depends only upon the dissimilar lower parts. Experiments have shown that if a large number of symbols in a user set contain cusps, segmentation on cusps results in a higher percentage of correct classifications.

Several methods were investigated for dissecting strokes into segments. The first technique tried depended upon detecting the relatively slow pen velocity in the vicinity of cusps. An inverse measurement of pen velocity was obtained by counting the number of tablet coordinates rejected in the pre-processing operation. Experimental data showed that the writing velocity differed between users and also for different symbols inscribed by the same user. Relatively slow velocity was observed for smooth portions of strokes as well as at cusps. In general the results of segmentation on velocity measurements were found to be unreliable.

The dissecting method used in the experimental recognizer locates cusps using geometrical measurements and is insensitive to pen velocity. The algorithm operates on the sequence of filtered coordinates which approximate a stroke. Cusps are isolated when the included angle between three successive points is less than a constant (normally set to 30°). Although cusps are reliably detected, sharp corners may or may not cause segmentation. Hence if a user desires to employ such one stroke symbols as a narrow V or N, he may need to include alternative descriptors in his dictionary.

*Experimental symbol recognizer*

### Multiple stroke symbols

A symbol may be composed of several strokes and one symbol can be a subset of others. Consequently an on-line recognition program must provide some means to detect symbol completion. One possible technique divides the writing surface into a grid and each symbol must start in a new space. This constraint may be acceptable when the data to be entered is in tabular form, but the technique is unsuitable for randomly placed symbols of varying sizes. A second method makes use of a tree-structured dictionary.[10,11] After a particular stroke has been classified, the dictionary is referred to for a list of permissible successor strokes. If the next stroke does not have an allowable identity and/or geometrical position, it is assumed to be assoicated with a new symbol. This technique has the advantage of reducing dictionary search time, a desirable feature when there are a number of entries for each symbol type. However, a poorly inscribed or positioned stroke may not correspond to an allowable successor and can abort the recognition process. In many of these abort instances the complete symbol contains sufficient information for correct classification.

In the author's recognition system classification is obtained from the best match of the complete input symbol with dictionary entries having the same number of strokes and segments. No attempt is made to identify the component segments of a multi-segment symbol. Hence it was necessary to devise a symbol completion algorithm which operates independently of the recognition process. The basic technique is to center a stroke or a subsymbol in a somewhat larger rectangle. If the next stroke does not enter this rectangle it is assumed to start a new symbol. This procedure automatically adjusts to varying symbol sizes.

The precise dimensions of the enclosing rectangle is determined by the aspect ratio of the stroke and in some cases the predecessor stroke is also a factor. If the height-to-width ratio R of the initial stroke is in the range of 1/3 to 3, the stroke is bordered on the top and bottom by m/2 and on the sides by m/4, where m is the maximum of the stroke width or height.

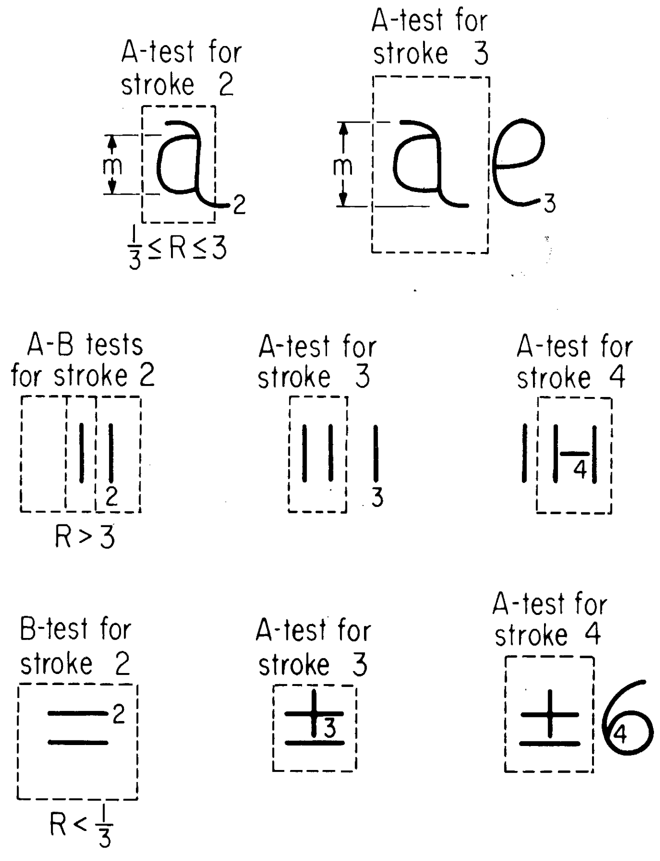


Figure 11—Symbol completion algorithm examples

If the second stroke enters this rectangle, the combined strokes are enclosed with the minimum rectangle plus the m/2 and m/4 borders, where m is now the maximum dimension of the symbol or subsymbol. The procedure is repeated until a new stroke fails to enter the rectangle. (See example at the top of Figure 11.)

When the initial stroke is tall and narrow with R > 3, two different enclosing rectangles are employed. Rectangle A is of the previously mentioned type and borders the top and bottom of the stroke by m/2 and the sides by m/4. If the 2nd stroke enters this rather narrow box, the two strokes are assumed to belong to the same symbol or subsymbol and the A test is repeated. Second strokes which do not enter the A-rectangle but have an R > 3 are tested to see if they enter the B-rectangle. The latter is actually a square of dimension 2m. When the B test has a positive result, the first two strokes are enclosed with an A-rectangle. If the 3rd stroke is completely within this rectangle, all three strokes are assumed to belong to the same symbol or subsymbol and the A-test is continued. However if the 3rd stroke is not enclosed by the A-rectangle, the first stroke is assumed to be a complete symbol and the 3rd stroke is treated as a possible second stroke for the next symbol. As shown in the middle of Figure 11, this feature of the symbol separation algorithm permits 1's to be more closely spaced than the vertical bars of an H. When the initial stroke is wide and short with R < 1/3, the B-test is applied to the second stroke. If the result is positive, the A-test is applied to subsequent strokes. (See example at the bottom of Figure 11.)

In addition to satisfying one of the spatial tests which have been mentioned, a component stroke of a symbol must be made within an interval of time (prescribed by the user) after the previous stroke. In this manner the program can detect the completion of an isolated symbol or the last symbol in a string.

## Segment spatial data

If two or more symbols in a user's set are formed with the same sequence of segments (e.g., the + — T or the n — h of Figure 12), the corresponding dictionary entries will contain identical contour vector sequences. To enable distinction, the program extracts and stores spatial information on the relative location of the center of component segments with respect to the center of a completed symbol. As shown in Figure 12, this relation is encoded as up (U), down (D), right (R), left (L), or coincident (C). When the spatial data for a particular segment in a symbol is needed for recognition, the user sets a bit in the corresponding segment

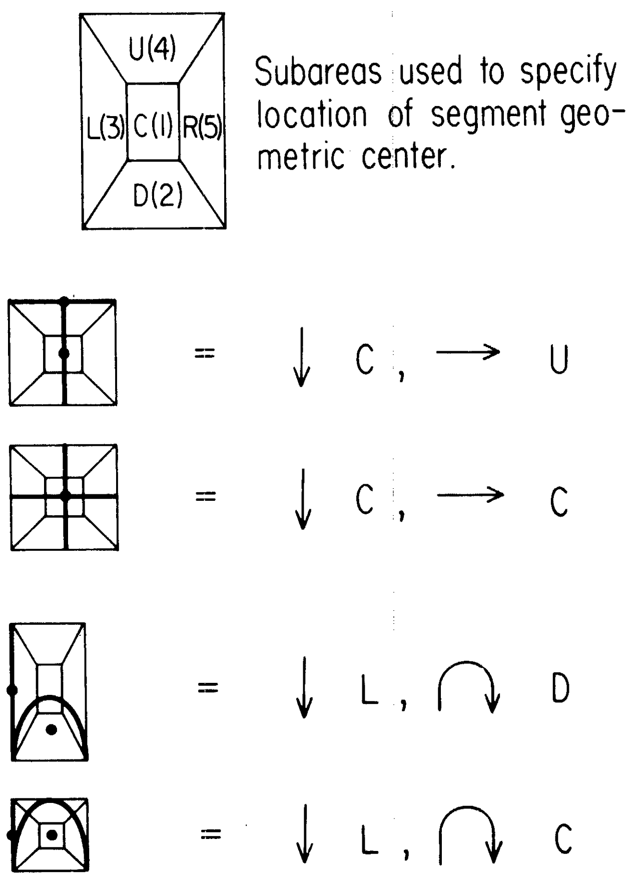

U(4)
L(3) C(1) R(5)
D(2)
Subareas used to specify location of segment geometric center.
= C, U
= C, C
= L, D
= L, C

Figure 12—Geometric position of segments

descriptor. The task of deciding which bits to set is made rather simple by a routine called TROUBLESHOOT which informs the user of all symbol pairs having a low mismatch. As indicated by an example set of symbols included in a later section, the segment spatial data is seldom required. Hence the recognizer has inherent tolerance to sloppy positioning of the component segments of most symbols.

*Dictionary building and testing*

The procedure for constructing a personalized dictionary is simple and fast. In this mode the user inscribes a symbol and its contour vector sequences appear on the CRT. In the experimental recongizer the CVS's are in numeric form, but dictionary construction could be simplified further by displaying each CVS as six connected vectors. The same symbol is repeated several times to see if there are any variations. The user then selects a representative sample and assigns an output code by pointing to one or two characters displayed on the CRT. (In the present arrangement the output code may correspond to a teletype character or a teletype character preceded by an ampersand.) The operation is repeated for other symbols until the dictionary contains the desired symbol set. At this stage in the procedure the dictionary entries are initialized not to use segment spatial data.

Next the recognizer is placed in a TROUBLESHOOT mode which computes the mismatch between every pair of symbols in the dictionary having the same number of strokes and segments. Symbol pairs with a low mismatch are displayed to the user. He may make his dictionary more robust by prescribing spatial differences, changing the form of symbols, or permuting stroke order. Or he may choose to ignore symbol pairs having low mismatch until poor recognition is actually experienced. Unwanted symbol descriptors are removed from the dictionary using the command DELETE followed by the corresponding output codes. The user can also save his dictionary on standard system files and retrieve a dictionary from these files.

Figure 13 shows the printing style used by the author

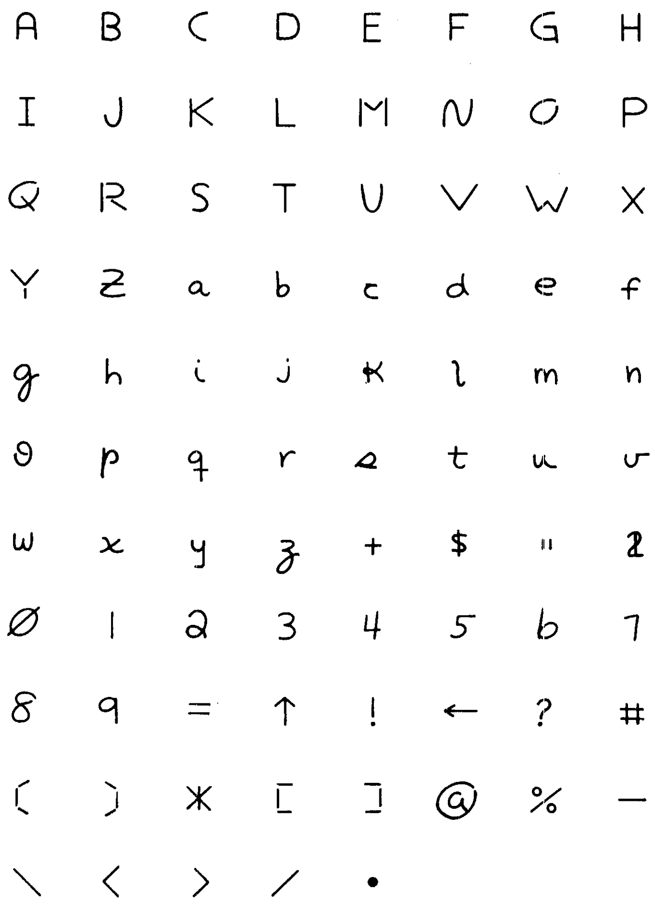


Figure 13—Test symbol set

```
4    04442222   05444444    H    03444444   05444444   01222222
A    01001344   01222222    I    01444444   04222222   42222222
D    03444444   41234566    K    43444444   04555555   02333333
M    03444444   01331344    R    03444444   04223566   02333333
P    03444444   44223566    (    04555555   03444444   02333333
Q    01654210   05333333    )    04333333   05444444   02555555
T    01444444   44222222    *    41444444   01555555   01333333
X    01333333   01555555    [    04666666   03444444   02222222
Y    04333111   02444444    ]    04222222   05444444   02666666
Z    01235522   01222222    B    03444444   04223456   12223566
&F   05654444   41222222    &Q   04654210   11444444   01222222
&I   01444322   01000000    @    01654210   15444322   01654210
&J   01444566   04000000    %    03654210   01555555   02654210
&K   03444444   01557333    &M   03444444   13001244   15001344
&T   05444322   01222222    F    03444444   04222222   41222222
&X   01234422   01555555    1    01444444
+    01444444   41222222    2    01245723
$    01643356   01444444    6    01442064
G    01654322   05222444    7    01223444
"    03444444   05444444    8    01643501
&    01245522   01444444    C    01654432
5    01422466   04222222    J    01444570
=    04222222   02222222    L    01444322
9    04654210   15444444    N    01013310
&A   03654210   15444321    O    01654210
&B   03444444   12012456    S    01553356
&C   04222222   11665322    U    01443100
&D   03653100   15444432    V    01333111
&G   04654210   11444011    W    01331311
&H   03444444   52001344    &L   01244432
&N   03444444   51001344    &O   01420632
&P   03444444   51001246    &V   01431022
&R   03444444   11000122    -    01222222
&S   01112456   12222222    \    01333333
&U   03443100   15444322    <    01555333
&V   03443100   15443100    >    01333555
&Y   04443100   11444566    /    01555555
&Z   04223466   12235711    &E   01216532
↑    01444444   04111333    •    01000000
|    01444444   05000000
←    01222222   03555333
3    04223566   12223566
?    01124554   02000000
#    03444444   05444444    04222222   02222222
E    03444444   04222222    01222222   02222222
```

Figure 14—Dictionary entries

for 85 different symbols. The set includes the ten numerals, the upper and lower case alphabetical characters, and 23 common teletype symbols. Lower case letters which are normally printed the same as upper case were changed to a cursive style. This was necessary because the experimental recognizer does not make use of relative size information.

A printout of the dictionary entries created by the author are shown in Figure 14. The output code selected for lower case alphabetics consists of the corresponding upper case symbol preceded by an ampersand. Each segment is represented by eight octal digits and the six on the right are used for the CVS. The 2nd digit from the left contains numbers 1 through 5 to indicate the location of the geometric center of the segment in a rectangle enclosing the complete symbol (See Figure 12). This geometrical relationship is required during recognition if the user has set the most significant bit of the first digit on the left. A "1" in the least significant bit of the 1st digit indicates that the segment continues a stroke and this information is used to partition the dictionary into subsets of symbols having the same number of strokes and segments. As the middle bit is currently unused, the first digit can be only a 0, 1, 4, or 5.

The TROUBLESHOOT parameter was set to list all symbol pairs having a mismatch of four or less. Figure 15 shows that the pairs C – <, L – ∫ U–V, and V–W each had a Lee distance of exactly four. The author decided to accept this level of mismatch for single segment symbols unless subsequent tests suggested a change of form. Figure 15 also lists ten pairs of multi-segment symbols having a total Lee distance of four or less. Seven of these pairs were made more robust by taking advantage of reliable differences in the relative position of component segments. Only one spatial bit was set in each symbol, thus allowing sloppy positioning of segments not needed for distinction. For example the second stroke of the f-T was chosen because it was felt that the horizontal bars of these two symbols would always be coincident and up respectively. However the first stroke of the f might at times be coincident and would then provide no spatial difference with the T. The SPATIAL DATA mode is called by typing SP and the number in the dictionary list of the desired symbol. A routine then automatically requests a no (N) or yes (Y) decision to set the spatial bit for each segment of the symbol.

Although the manual setting of spatial bits is greatly facilitated by the TROUBLESHOOT mode, the procedure does require familiarity with the fundamental principles of the recognizer. The task could be accomplished automatically, but would require the user to provide additional input samples. The F-I pair in the

```
D  03444444    41234566
      0           2
P  03444444    44223566                      C  01654432
                                                    4
I  01444444    44222222                      <  01555333
      3           0
&F05654444    41222222                       L  01444322
                                                    4
T  01444444    44222222                      &L01244432
      0           0
+  01444444    41222222                      U  01443100
                                                    4
&F05654444    41222222                       V  01333111
      3           0
+  01444444    41222222                      V  01333111
                                                    4
&A03654210    15444321                       W  01331311
      0           4
&G04654210    11444011

&H03444444    52001344
      0           0
&N03444444    51001344
                              F  03444444    04222222    41222222
&H03444444    52001344              0           0           0
      0           3           I  01444444    04222222    42222222
&P03444444    51001246
                              K  43444444    04555555    02333333
&N03444444    51001344              0           0           0
      0           3           *  41444444    01555555    01333333
&P03444444    51001246
```

Figure 15—TROUBLESHOOT list

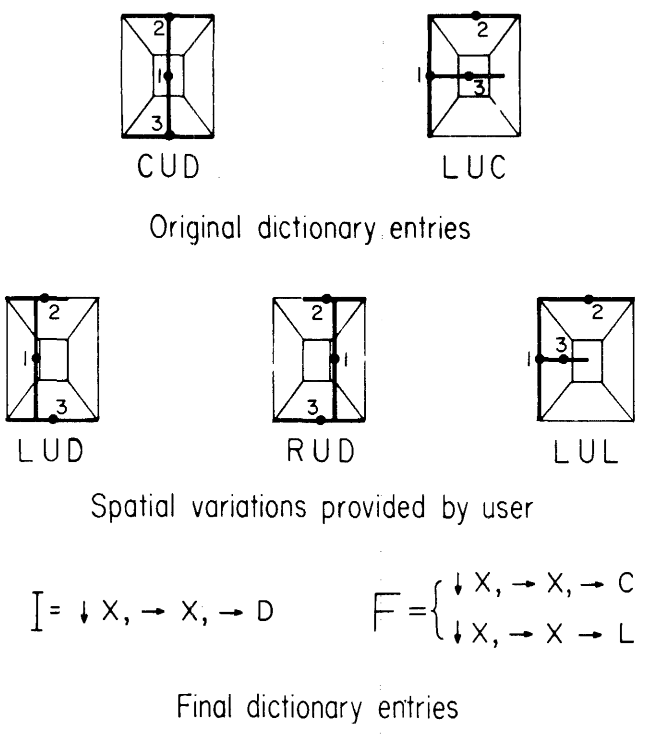


Figure 16—Automatic setting of spatial bits

```
T   Ø1444444   Ø4222232   MIS-MATCH: 1
H   Ø3444444   Ø5444444   Ø1122222   MIS-MATCH: 1
E   Ø3444444   Ø4222223   Ø1212222   Ø2122232   MIS-MATCH: 4
Q   Ø1654210   Ø2333333   MIS-MATCH: Ø
U   Ø1442100   MIS-MATCH: 1
I   Ø3444444   Ø4222221   Ø2232223   MIS-MATCH: 3
C   Ø1655422   MIS-MATCH: 2
K   Ø3444444   Ø4555555   Ø2333333   MIS-MATCH: Ø
F   Ø3444440   Ø4222223   Ø1222233   MIS-MATCH: 7
O   Ø1654210   MIS-MATCH: Ø
X   Ø1434333   Ø1555555   MIS-MATCH: 2
J   Ø1444570   MIS-MATCH: Ø
U   Ø1442100   MIS-MATCH: 1
M   Ø3444444   Ø1311444   MIS-MATCH: 3
P   Ø3444444   Ø4234566   MIS-MATCH: 2
S   Ø1642356   MIS-MATCH: 2
O   Ø1654211   MIS-MATCH: 1
V   Ø1342111   MIS-MATCH: 2
E   Ø3444444   Ø4222222   Ø1222222   Ø2222222   MIS-MATCH: Ø
R   Ø3444444   Ø4224566   Ø2333332   MIS-MATCH: 2
T   Ø5444445   Ø4222222   MIS-MATCH: 1
H   Ø3444444   Ø5444444   Ø1222222   MIS-MATCH: Ø
E   Ø3444440   Ø4222222   Ø1222223   Ø2222222   MIS-MATCH: 5
L   Ø1444322   MIS-MATCH: Ø
A   Ø1001344   Ø1221222   MIS-MATCH: 1
Z   Ø1245522   Ø1222222   MIS-MATCH: 1
Y   Ø4332111   Ø2444444   MIS-MATCH: 1
B   Ø3444434   Ø4223556   12223566   MIS-MATCH: 2
R   Ø3444444   Ø4234566   Ø2333333   MIS-MATCH: 2
O   Ø1654210   MIS-MATCH: Ø
W   Ø1321310   MIS-MATCH: 2
N   Ø1024311   MIS-MATCH: 3
D   Ø3444444   Ø1234566   MIS-MATCH: Ø
O   Ø1654210   MIS-MATCH: Ø
G   Ø1554322   Ø5224444   MIS-MATCH: 3
```

Figure 17—TEST on upper-case letters

```
&T  Ø1444432   Ø1222222   MIS-MATCH: 2
&H  Ø3444444   12001444   MIS-MATCH: 1
&E  Ø1216532   MIS-MATCH: Ø
&Q  Ø4655321   15444444   Ø1222222   MIS-MATCH: 4
&U  Ø3442100   15444222   MIS-MATCH: 2
&I  Ø1444322   Ø4000000   MIS-MATCH: Ø
&C  Ø4222222   11665322   MIS-MATCH: Ø
&K  Ø3444440   Ø1556133   MIS-MATCH: 7
&F  Ø1654444   Ø1121222   MIS-MATCH: 2
&O  Ø1310632   MIS-MATCH: 2
X   Ø1344332   Ø1555555   MIS-MATCH: 3
&J  Ø1444566   Ø4000000   MIS-MATCH: Ø
&U  Ø3443100   15444322   MIS-MATCH: Ø
&M  Ø3444444   13001344   15011444   MIS-MATCH: 3
&P  Ø3444444   11001356   MIS-MATCH: 2
&S  Ø1012456   12222222   MIS-MATCH: 1
&O  Ø1410742   MIS-MATCH: 3
&V  Ø1431022   MIS-MATCH: Ø
&E  Ø1206532   MIS-MATCH: 1
&R  Ø3333444   11000112   MIS-MATCH: 4
&T  Ø1444422   Ø4222222   MIS-MATCH: 1
&H  Ø3444444   12011444   MIS-MATCH: 2
&E  Ø1216532   MIS-MATCH: Ø
&L  Ø1244432   MIS-MATCH: Ø
&A  Ø5654211   15543221   MIS-MATCH: 4
&Z  Ø4223455   11345012   MIS-MATCH: 6
&Y  Ø4543211   11444457   MIS-MATCH: 7
&B  Ø3444444   15001356   MIS-MATCH: 3
&R  Ø3344444   11000122   MIS-MATCH: 1
&O  Ø1420742   MIS-MATCH: 2
&W  Ø3432110   15442100   MIS-MATCH: 4
&N  Ø3444444   11011444   MIS-MATCH: 2
&D  Ø5653100   15444432   MIS-MATCH: Ø
&O  Ø1310642   MIS-MATCH: 3
&G  Ø4654210   11445011   MIS-MATCH: 1
```

Figure 18—TEST on lower-case letters

upper part of Figure 16 will be used as an example. Assume that additional training samples produce the spatial variations shown in the middle of the figure. The spatial bit routine would determine from all five samples that the third stroke provides reliable distinction, and that two dictionary entries are required for the F. (X means that the spatial bit is not set.) In contrast to other training methods, the user is required to provide samples only for symbol pairs having low mismatch. The minimal use of spatial information results in a recognizer which is very tolerant to inaccurate positioning of the component strokes of most symbols.

A TEST mode allows the user to further evaluate his dictionary. He simply draws a sequence of symbols which are separated from each other by at least one quarter of the maximum symbol dimension. The symbol separation algorithm determines when a symbol has been completed, and the recognition routine guesses the identity of the symbol on the basis of lowest mismatch. Mismatch calculations are made between the input symbol and all dictionary entries having the same number of strokes and segments. If two or more symbols have the lowest mismatch, the first one encountered in the dictionary search is chosen. Dictionary entries in which a spatial bit is set require a specific location for the corresponding input segment.

Figures 17 and 18 show some test results obtained on the author's symbol set. The displayed or teletype output from the TEST mode includes the dictionary entry guessed, the segment descriptors for the input symbol, and the mismatch between the former and the latter. In this particular test the phrase "the quick fox jumps over the lazy brown dog" was inscribed in upper and lower case letters. Out of a total of 70 inscribed symbols, the only error was a lower case x which misread as an upper case X.

## CONCLUSION

The author's dictionary entries (see Figure 13) were also used to classify the distorted one and two stroke letters of Figure 19. Except for a T which misread as a t, all of the characters in this figure were classified correctly (the amount of mismatch is shown below each symbol). Although previously developed on-line recognition schemes also are capable of recognizing distorted symbols, they require the user to provide a large number of training samples. The nearest prototype technique described in this paper performs the task with a single dictionary entry per symbol.

The symbol recognizer has been used by many different people and all have found it enjoyable to operate. In one experiment three subjects were asked to construct personalized dictionaries consisting of the numerals, the upper case letters, and the lower case letters which differed from upper case. Each of these persons adjusted to using the tablet and CRT display within 15 minutes and then took about a minute to make each dictionary entry. As the automatic means for setting spatial bits has not been implemented, the subjects were given brief instructions on the manual procedure. The operation itself took about 15 minutes.

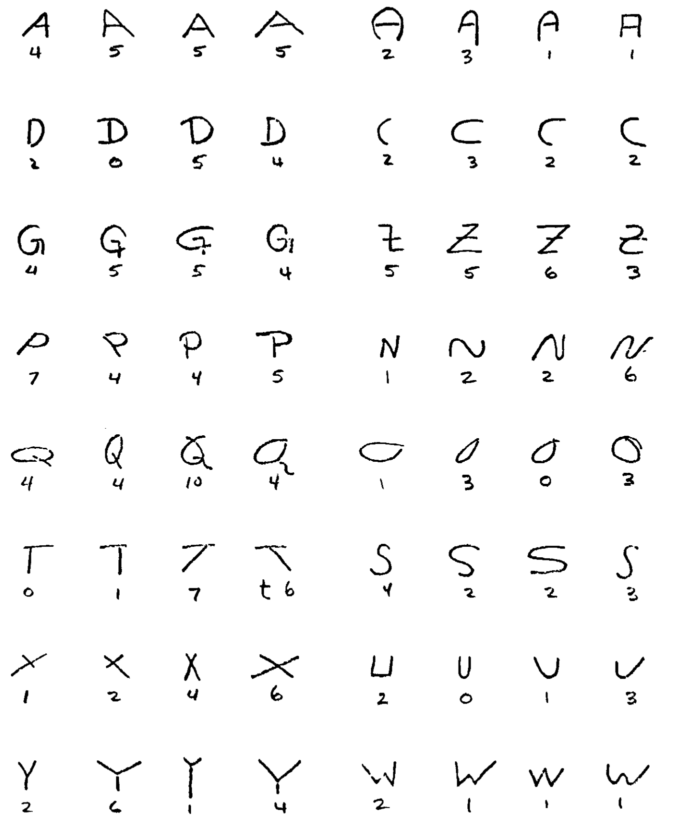


Figure 19—Intentionally distorted symbols

After their dictionaries were constructed the subjects were asked to write the complete alphabets and the phrase "the quick fox jumps over the lazy brown dog" in upper and lower case. From this test of 132 symbols a user typically had two to five misreads. With additional experience and very slight refinements of dictionaries, all subjects obtained recognition rates in excess of 98 percent. An error rate of 5 percent is generally considered acceptable in on-line systems, because each character can be classified, displayed, and corrected immediately by the writer if it is wrong.

The compiled program for the symbol recognizer requires approximately 9K 24-bit words of memory. On the average an additional 4 words are required for each dictionary entry. Owing to the simplicity of the mismatch calculations and the high speed of the SDS 940 computer, the recognizer can easily accommodate normal writing rates of symbols from a set of 100.

The CVS signature and Lee metric is a fundamental technique for measuring the similarity of two arbitrary curves, and can be applied to a wide spectrum of pattern classification problems. The author is currently investigating the usefulness of the method for machine recognition of cursive writing in lower case letters. Preliminary results from this research are contained in Reference 9.

## ACKNOWLEDGMENTS

The author wishes to express his gratitude to the many people of Project Genie whose work has produced the TSS used to implement this research. Chacko Neroth and Ken Pier gave valuable assistance in early experimental work on the basic algorithms, Barry Borgeson provided software for the tablet and display, and Bo Lewendal wrote much of the program incorporated in the experimental recognizer.

Support for this work has come from the Advanced Research Projects Agency of the Office of Secretary of Defense under Contract No. SD-185.

## REFERENCES

1 D B DEVOE

*Alternatives to handprinting in the manual entry of data*
IEEE Trans of Human Factors in Electronics Vol 8 No 1
March 1967 21-32

2 T G WILLIAMS C H FRYE
*An instruction application of computer graphics*
Educational Tech 5–10 June 15 1968

3 G D HORNBUCKLE
*The computer graphics user machine interface*
IEEE Trans of Human Factors in Electronics Vol 8 No 1
March 1967 17-20

4 J H MUNSON
*Experiments in the recognition of hand-printed text: Part I character recognition*
Proc FJCC 1968 1125-1138

5 G NAGY
*State of the art in pattern recognition*
Proc IEEE Vol 56 No 5 May 1968 836-862

6 R O DUDA P E HART
*Experiments in the recognition of hand-printed text: Part II—context analysis*
Proc FJCC 1968 1139-1149

7 M R DAVIS T O ELLIES
*The RAND tablet: A man-machine graphical communication device*
Proc FJCC 1964 325-331

8 B W LAMPSON W W LICHTENBERGER
M W PIRTLE
*A user machine in a time-sharing system*
Proc IEEE Vol 54 No 12 Dec 1966 1766-1774

9 G M MILLER
*On-line computer recognition of handwritten symbols*
Elec Engrg Dept Univ of Wis 1969 PhD Dissertation

10 G F GRONER
*Real-time recognition of hand-printed text*
Proc FJCC 1966 591-601

11 M I BERNSTEIN T G WILLIAMS
*A two-dimensional programming system*
I F I P Congress Edinburgh Scotland Aug 5-10
1968 C84-C89

12 J H MUNSON
*Some views on pattern-recognition methodology*
Internat Conf of Methodologies of Pattern Recognition
Univ of Hawaii Honolulu Jan 24-26 1969

13 H FREEMEN
*On the encoding of arbitrary geometric configurations*
IEEE Trans of EC Vol 10 No 2 June 1961 260-268

14 J FEDER H FREEMEN
*Digital curve matching using a contour correlation algorithm*
Proc IEEE int Conf March 1966 69-85

15 E R BERLEKAMP
*Algebraic coding theory*
McGraw-Hill Book Co 1968 Chapt 8 204-205

# Common file organization techniques compared

by NED CHAPIN

*InfoSci Inc.*
Menlo Park, California

## INTRODUCTION

In order to make a comparison of file organization techniques, concurrence is needed on terminology. To that end, this introduction offers some definition of terms. Unfortunately, many of these terms do not have universally accepted definitions. A general definition of terms can be found elsewhere.[5]

In offering definitions of terms, this paper does not suggest that those who give different definitions are wrong. On the contrary, the differences in definition that exist reflect in part imperfect communication among people in the field, and in part real differences in the concerns of the people in the field. Hopefully, papers such as this one will help improve communication. But the differences in concern will continue to exist, and to spawn both new differences and new terms.

As used in this paper, the term "file organization" is not synonymous with file structure, data structure, data base, data organization, or data management. A file organization is viewed as a way of putting together the components of a file. "File structure" is viewed as synonymous with file organization, but is not used in order to help distinguish it from "data structure." A "data structure" is a more general term than file organization, since a file is viewed as but one general organization of data. Some people use the term data structure to refer only to vertical relationships among data. "Data organization" is viewed as synonymous with data structure. A data base is viewed here as a group of files or alternatively as a controlled aggregation of data which can be regarded as organized into files.

The term "data management" is used with a variety of meanings in the field. Sometimes it is narrowly used to refer to movement and formatting of data to and from internal storage, and the supporting software. Sometimes in a broader sense it also refers to the identification of data and procedures to maintain the integrity and security of the data. At other times, the term is used also to refer to file organization. In a very broad sense, it refers also to the maintenance of files, the handling of inquiries, and the preparation of reports.

These definitions raise questions about the definition of the vertical and horizontal organization of data. Looking first vertically, this paper views a file as an arbitrary but usually homogeneous but not exhaustive aggregation of records. Records are collections of data all of which share some attribute in common, usually the name of a thing the data are about. For example, a record of employee job attendance might contain data about number of days worked, number of days absent, the usual work station, the parking lot location, the home address, the home telephone, the usual days of the week absent, and the like. When these data are drawn together and grouped in terms of the identification of the employee (such as by employee identification number), the individual groupings thus formed are here viewed as records. The components of the record are data items (usually fields), as diagrammed in Figure 1.

The definition of a record implies no specific ordering of the data items. The definition of the file implies no ordering of the records within the file. By ordering is meant the application of a collating sequence or pattern template to data items at a uniform level in the vertical hierarchy of data. When records are ordered,

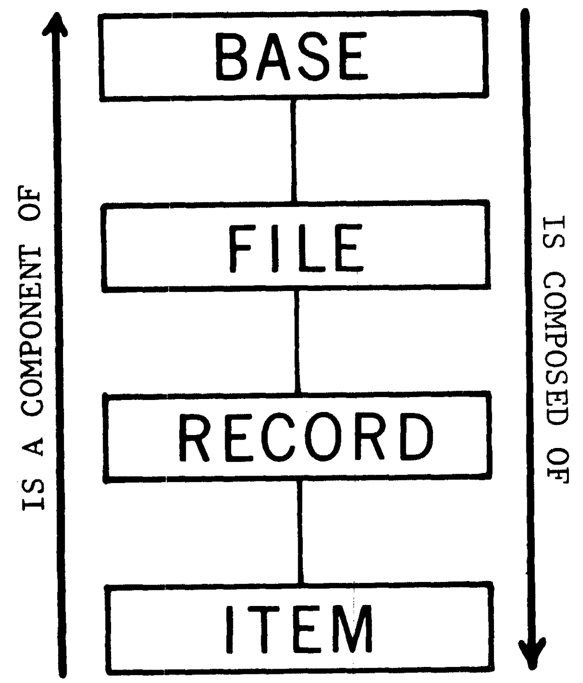


Figure 1—Condensed diagram of the vertical hierarchy of data

the data items used for the ordering are referred to here collectively as the key. For example, the records in the attendance file just cited might be ordered using an ascending numeric collating sequence with the employee identification numbers serving as the key.

The horizontal organizations of data reflected in this paper require definitions of table, tree, string, and list. A "table" is a series of pairs of data items, which are the argument and the function. The table by its form permits the table user to establish by inference a relationship between a particular argument and its associated function. A telephone book and a statement of tax rates are examples of tables.

Three important tables for the comparison of file organizations are indexes, directories, and tables of contents. An "index" has the arguments in a specific order but the function which may consist of multiple data items may be in order. By contrast, a "table of contents" cites the functions in a specific order but leaves the arguments in any order. "Directories" may have the arguments or functions or both ordered in any manner. For this reason, the term directory serves as a general term covering in practice both indexes and tables of content.

A "tree" can be used to represent vertical relationships among data.[4] A tree may also be used for horizontal organization of data, as shown in Figure 2. For

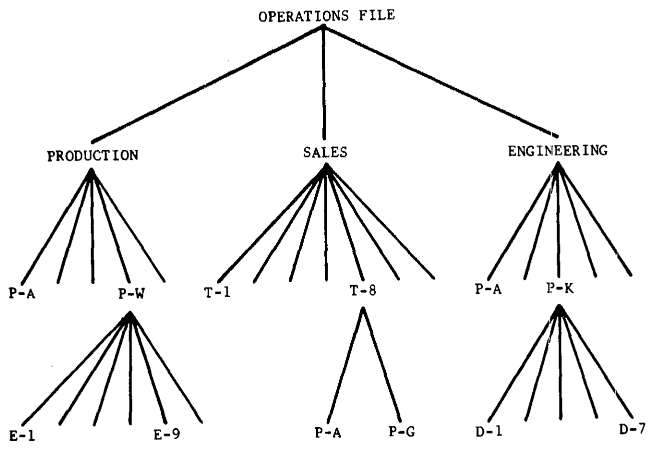


Figure 2—A partial representation of a tree as a horizontal organization for a file

example data about a firm's operations might be broken into divisions such as production, sales, engineering, and the like. These divisions in turn can be broken into subdivisions. For example, sales might be broken into territories, and production into the product categories. Engineering might incorporate new product categories currently not in production, as well as those in production. These categories can in turn be broken still further. Thus in production they might be broken by production equipment or in terms of a bill of material. In sales they might be broken down into products or into salesmen. In summary the term tree gets its name from the graphic representation of the processes of subdividing.

By contrast, a string organization is viewed as a series of things, one after the other, where the elements composing the series are similar. Examples of strings are series of characters, of digits, of names, or of numbers.

A "list" is viewed as a series of records or data items, each accompanied by one or more pointers to other elements in the series. These pointers are here termed "links" and are themselves data items. Some people prefer the term "chain" to refer to a list.

Irrespective of vertical or horizontal aspects of the file organization, a file may exhibit a simple or a compound organization. A "simple" organization has only one major structural pattern. A "compound" organization has two or more distinct and different structural patterns which taken together comprise the file organization.

### *Classifications*

The number of people in the field have proposed classifications of file organization. A brief review of some of these will serve as a basis for selecting one for use in making comparison here.

A team headed by Anthony J. Dowkart has offered an extensive basis for comparison.[9] In summary, this basis is: the data definition provided, the facilities for file creation and maintenance, the retrieval mechanism, the processing procedures, the output characteristics, and the operating environment. This basis of classification is concerned not with file organization alone, but also with data management in the broad sense. Looking at the matter of file creation and maintenance, and of data definition, the classification bases suggested are performance oriented, rather than structure or pattern oriented.

Richard G. Canning has suggested classifying file organization into two general classes based upon type and upon structure.[3] Within type he proposes recognizing sequential, indexed, and chained files. Within structure, he proposes recognizing linear, hierarchical, and involute files. These classifications are more structure and pattern oriented than those just cited, but they lack a consistently applied, obvious basis.

Minker and Sable in reviewing data management systems suggested a basis of classification as user language, file structure, system processing capability, and user interface.[13] This again shares the same general user basis cited previously. Looking more particularly at the basis identified as file structure, Minker and Sable suggested classifying on the basis of the implementing storage media (such as tape or disk) and the variety of field and record lengths permitted. Among those that permit greater variety and which are disk based, Minker and Sable suggested a classification of indexed, tree-ordered, and linked, or chained. These suggestions share many of the features of those of Canning as noted earlier.

David Lefkovitz has suggested a classification of file organization based upon a combination of the hardware and software components utilized to implement the file.[12] These he viewed from a functional point of view, particularly with regard to the retrieval process. Thus a file organization may be classified on the basis of which software-hardware components it utilizes and in what way. For example, does it use a directory, does it use a randomizing or a tree approach? If it uses a tree approach, does it use a fixed length key or a variable length key? And so on. Such a basis of classification results in a very large number of possible classes. In a sense, each non-identical existing file organization becomes a separate classification.

Ned Chapin has suggested a classification scheme based fundamentally upon the way of indicating association at a given vertical level within a file.[4] At one extreme he placed the attributed organization which provides explicit identification with the data at some given level. This obviates the necessity for providing a means of association below this level. At another extreme, he placed the linked or list organization, where each data element at a given level incorporates a specific indication of association. Two varieties of this he singled out for particular attention: the complex ring which is a complex list that forms closed loops, and the muble or multiple double-linked list which provides two or more links. At another extreme, he placed the hierarchical organization, which provides a tree-like association on a horizontal basis. Finally, at another extreme, he placed the positional organization. This provides association in terms of placement in relation to other data, at a given vertical level. Thus, field A is always known to precede field B, and field B is always known to precede field C, and all three fields are always present in a record. Hence, values from the third field position have a known identification and association.

The Chapin classification utilizes an important feature of the way people think about data, as its basis for classification. As such, it avoids the mixed base problems inherent in the other classification schemes it reviewed, without the gaps or holes characteristic of the other systems.

This classification approach lends itself to a graphic representation, as diagrammed in Figure 3. The diagram uses time as the left to right distance, but not in strict scale units.[7] The vertices or nodes are the identity of data. The solid arcs or lines are the sequence of the active (pointed to) data. Vertically, the diagram has two parts, an upper or demand (D) part, and a lower or supply part. A perfect match of the file organization to the demands upon it occurs when the data (indicated by broken lines) demanded and supplied occur at the same time.

*Characteristics*

The point is well taken that users by and large are

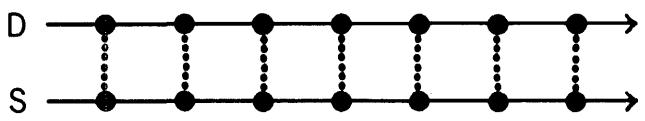


Figure 3—A graphic representation of associations showing the ideal pattern for a file organization

unconcerned with the classification of a particular file organization technique. They are concerned with the functional characteristics of the file organization technique in action. Some of these of course are hardware and software dependent. But within those bounds, they are determined largely by the file organization itself. Among the common characteristics are the speed and basis of access, the use of storage capacity, the ease of maintenance (for insertions, alterations, and deletions), and the extent of software support available.

The speed and basis of access is fundamentally affected by the association provided in the file organization because access uses the association for its realization. The hardware, the software, and the association together set the limits. The basis of access may be by attribute, by value, or by property as has been pointed out elsewhere.[4]

The use of storage capacity reflects two aspects of file organization, each of which in turn rests upon the basis of association. One aspect is that compound organizations commonly use more storage capacity than do simple ones. Another is that hardware and software factors also affect the use of storage, given the file organization.

The procedures, the convenience, and the time required for maintenance operations, such as insertion, deletion, and alteration of data in a file, depend obviously upon the hardware and software used. But they also depend importantly upon the association provided by the file organization, since maintenance involves access, but is more than access. Common maintenance practice is not always a corollary of the features of the file organization.

The extent of software support available is a very significant determinant of the degree to which people are willing to use a file organization. Even if it be theoretically attractive, a file organization unsupported by software is in practice ignored in favor of anything that is supported by debugged software.

*Common techniques*

### Techniques covered

The most common file organization techniques are those proselytized and supported with software by the computer vendors. These are normally part of the operating system and are accessible to anyone who programs in the symbolic language for a particular computer. Some of them are available to users of higher level languages such as COBOL and FORTRAN.

Less commonly used are the file organization techniques supported by software available from the computer vendors but not provided normally as part of the operating systems. These usually take the form of "packages" capable of a variety of functions.

A third category are the file organization techniques available in the software market from independent suppliers of software. None of these are as common as those available in the first category, but some are as common as some in the second category.

For contrast, this paper looks also at the extensions to COBOL proposed to CODASYL in the area of file organization techniques.

### Vendor supported techniques

Historically the oldest, the most popular, and by far the most common, is the strict sequential file organization. The strict sequential is a positional organized file commonly consisting of ordered records which are themselves positionally organized.[4,10] As such, its use of storage is the most economical of all. It is a simple, not a compound organization.

The strict sequential enjoys a rapid next-record access by attribute, but a slow random access by attribute, as diagrammed in Figure 4. That is, as long as the sequence in which access is demanded conforms to the sequence in which the file was sorted, access is rapid unless the number of records to be passed over is large. Unfortunately, access is often desired on some other key. This requires first a reordering of the file which involves a time-consuming sorting operation, or an exhaustive search of the file. Even with this sorting

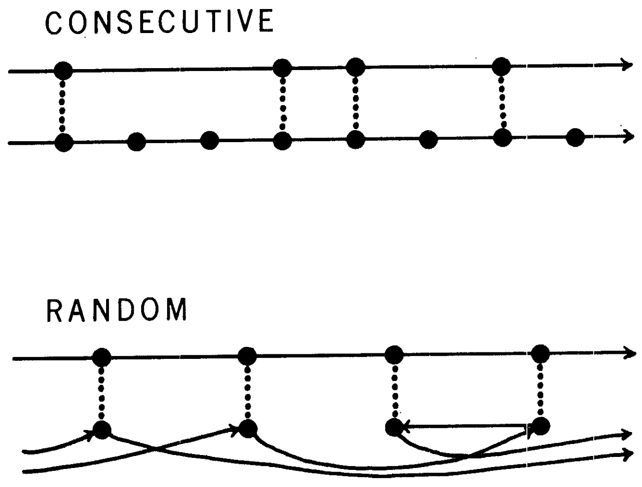


Figure 4—Diagram of the strict sequential file organization

operation, access by value and by property involve search.

Maintenance for sequential files is logically straightforward, but slow. It requires typically a complete passage of the file with a complete copying of it. Each record must be read and written in order to do maintenance on the file. Because of this, insertions and deletions are easily accomplished. Alterations are also simple as long as the typical fixed length restrictions on field sizes is observed. Where variable length fields are permitted alterations become a little more complex but are still logically straightforward.

Software support for sequential organization is extremely good. Its popularity is attested by Table I. It is the most widely supported of all the file organization techniques.

The indexed sequential is a compound file organization technique, historically younger than the strict sequential.[4,10] This too is a positional organization. The main file is a strict sequential file. With it is a sequential organized index using the same key. Sometimes indexes to indexes are provided depending upon the size of the main file and the storage space available.

Random access for the indexed sequential file is superior in speed to the strict sequential because the index search requires less time than a search of the main file. From the index the location of the desired record can be found and the record then accessed without search. But for a next-record access, the same procedure usually is required, which slows such access (see Figure 5). Access by attribute, by value, and by property follow the same pattern as for the sequential organized file.

The use of storage space for the indexed sequential is larger because of the additional space required for the indexes. An added inefficiency in the use of storage space is the typical requirement for overflow areas to permit insertions in the main files. This overflow may amount to as much as a third to a half more space for the main file, although typically this can be held to about one-tenth more space.

TABLE I—Summary of the file organization techniques supported by the eight largest computer vendors

| *Strict Sequential* | *Indexed Sequential* | *Direct or Random* |
|---|---|---|
| IBM | IBM | IBM |
| RCA | RCA | RCA |
| CDC | CDC | UNIVAC |
| UNIVAC | UNIVAC | NCR |
| Burroughs | NCR | GE |
| NCR | Honeywell | Honeywell |
| GE | | |
| Honeywell | | |

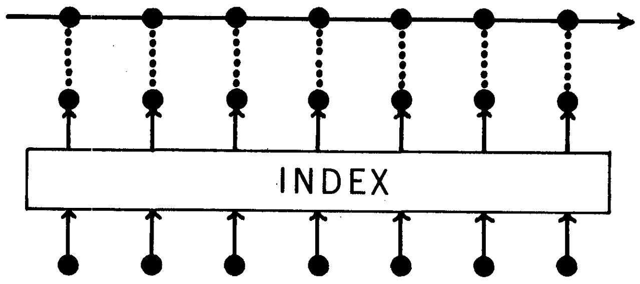


Figure 5—Diagram of the indexed sequential file organization

The maintenance of the indexed sequential file differs considerably from that for strict sequential. Maintenance does not require rewriting the entire file; only those specific records in the file that are altered are rewritten back into their same places. This saving in maintenance time can be more than offset by other factors.

An insertion in an indexed sequential file requires that adjustments be made to the index and to the main file. The inserted record typically must be written in the main area displacing a record into the overflow area. Links are inserted if more than one such overflow occurs in a given area. By contrast, deletion is more simple. The record to be deleted is simply marked for deletion but is not physically deleted from the file nor from the indexes. Periodically, the entire file is rewritten in order to eliminate the accumulated deletions, to pull the insertions into the main sequence, to reapportion the overflow areas, and to clean the index. In sum, whether or not the maintenance time for an indexed sequential file exceeds that for a strict sequential file depends upon the volume of insertions and alterations. For low to moderate volume, the strict sequential is usually slower over-all. An indexed sequential suffers from the same single-key limitations as the strict sequential.

The software support for indexed sequential generally is good. The software operates more slowly per random access than for strict sequential because of the decreased buffering possible.

The direct or random file organization is also a positional organization.[4,10] It is like strict sequential in that it is simple, not compound. The direct or random file organization is a variation of the strict sequential. It uses a transformation of the key. Whatever the key would be is passed through an algorithm to calculate a position in storage. Because of the possible occurrence of multiple records having the same key, or of closely spaced keys, provision is made in the algorithm to handle some conditions. One is to place or find a record when its transformed key is the same as another transformed key. This can be handled by links and overflow areas, or by shifting records to maintain a sequence in order to restrict the search domain. Another is to set up the initial spacing of records in the file to permit room for the later insertions. The amount of storage space allocated for this purpose is usually not less than that allowed for overflow areas in the case of an indexed sequential file.

The random access provided by the direct or random file organization is slightly faster than that for an indexed sequential organized file, since no index reference is needed. But for next-record access, it is slower because the transformed key order is not the same as the ordinary key. Hence, every access is a random access, as diagrammed in Figure 6. The access basis is the same as noted earlier for the positional organized files. Also, only one key can be used, as noted earlier.

The use of storage space for the direct or random file organization is about as efficient as that for the indexed sequential, and is less efficient than for the strict sequential. This is because of the voids that must be left in the spacing of the records to accommodate inserts, and the use of overflow areas. No space is needed for an index.

The maintenance for a direct or random organized file resembles the indexed sequential more than the strict sequential. This may also extend to alterations and deletions. For insertions, no index need to be adjusted. If the record to be inserted must go into a place that is already occupied (that is, the transformed key is a duplicate of an already existing transformed key) then provision must be made for moving records or for use of overflow area and links.

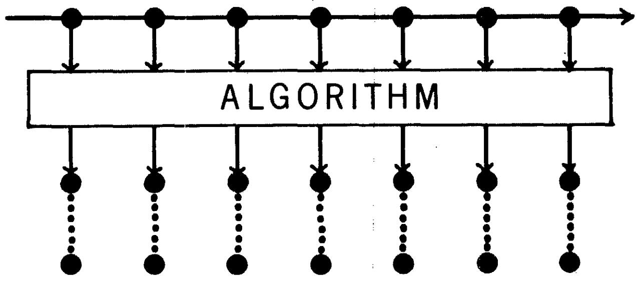


Figure 6—Diagram of the direct or random file organization

The software support for the direct or random file organization is less troublesome and less burdensome than that for the indexed sequential. Also, less supporting software is needed to accomplish the job. The user does not even need to rely upon manufacture provided software but can make do by providing his own algorithm for key transformations and by using a strict sequential file organization. Many vendors have been supplying this software for a longer period of time than they have supplied indexed sequential software.

Another type of common file organization technique available from the computer vendors and incorporated as a normal part of their operating systems is the partitioned file organization.[4,10] This is a hierarchical file organization. But it is normally not accessible to the programmer even though it is utilized routinely by the operating system for its own functions such as program libraries. Typically, the hierarchical file organizations are compound because they require directories and sometimes even hierarchies of directories to maintain association and provide access. These directories usually include one that is of the table of contents type.

Access by attribute is the most common. The speed of access depends mostly upon the size and number of directories used (see Figure 7). Maintenance is usually done by making deletions by altering only the directories. Insertions are entered in the directories and the new data placed in any available space. Alterations are often treated as combined deletions and insertions.

The software support is usually inadequate to enable the use of the partitioned file organization by programmers in their own programs. The organization becomes increasing uneconomically of storage space as deletions accumulate. To eliminate them requires rewriting the entire file and recreating the directories,

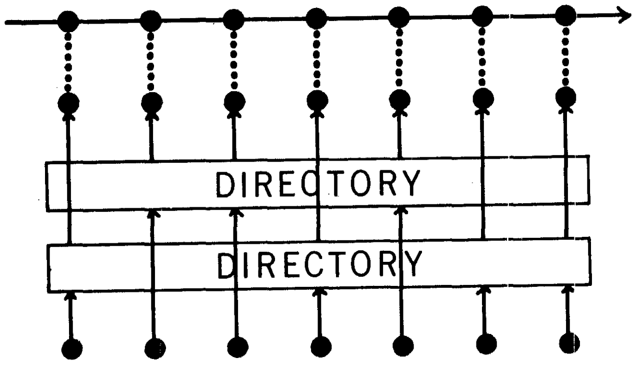


Figure 7—Diagram of the partitioned file organization

TABLE II—Summary of selected vendor augmentations

| *Strict Sequential* | *Indexed Sequential* | *Direct or Random* | *Other Techniques* |
|---|---|---|---|
| GIS | GIS | GIS | IDS (ring) |
| FORTE | UNIMS | FORTE | FORTE (list) |
| MARS | FORTE | | |
| | MARS | | |
| | UL/1 | | |

an operation equivalent to that needed for the indexed sequential file organization.

## Vendor augmentation

Computer vendors over the years have made a number of augmentations and elaborations of the implementation of file organizations just compared. The best known of these are listed in Table II.

One of these has been IBM's GIS (Generalized Information System).[2] This elaboration provides a number of features that add greatly to the power and convenience available to the user. Underlying it are the two positional organized file organizations, the strict sequential and direct or random. The use of indexed sequential is optional depending upon the scope of the GIS implemented. GIS is a free-standing package, not an extension of COBOL, but GIS can be used with COBOL.

The access for the GIS is slower because of the additional software. But that software yields greater convenience of user access by reducing programming effort to file and retrieve data. The use of storage space is but little more extensive, ignoring the space for the additional software. Maintenance follows the usual procedures but is more convenient from the user's point of view because he does not need to write all of the programs for doing it. The software support is comprehensive.

The Integrated Data Store (IDS) is available from General Electric,[1] and is similar to the General Motors Associative Programming Language. IDS offers a complex ring file organization where the number of links possible at the record level in the file may be made as extensive as the user desires. In practice, it is used most often as an extension of COBOL.

Access by attribute beyond the first access is slightly facilitated because of the links. Access by property is much facilitated as a practical matter because of the links which provide quick reference to the records with related contents. The use of storage space is greater than for a strict sequential organization because of the space occupied by the links. Since in practice, directories are used to locate or serve as pointers to rings, a little additional storage is also needed for them.

Although insertions, deletions, and alterations are handled by the software, the procedures are considerably more complicated for IDS than for the positional organized file. This is because of the need to adjust the links whenever insertions and deletions are made. If the insertion cannot be made physically nearby, then subsequent accesses following the links are slowed. This maintenance problem compounds as the number of links to be adjusted increases. The software support available for IDS is comprehensive and has been extensively tested in use.

The UNIMS (Univac Information Management System) is available from the Univac Division of Sperry Rand. It offers a modified indexed sequential file organization in a package of software, in a similar manner to that noted earlier for GIS and IDS. It too can serve as an extension to COBOL.

The access and maintenance for UNIMS are similar in character to that noted earlier for indexed sequential files. But to the user the procedures appear easier because of the assistance provided by the software. UNIMS uses little more storage space than the indexed sequential noted earlier. The software support is comprehensive.

The UL/1 (User Language/1) from RCA offers a more convenient language for the handling of access, maintenance, and reports from files than the usual programming languages. As such it has similar objectives to GIS noted earlier. UL/1 uses a modified indexed sequential file organization in a way that gives the appearance of a hierarchical file organization.[11] The characteristics of this software system were still fluid at the time of this paper.

FORTE is available from Burroughs Corporation. It provides unordered (sequential), indexed sequential, random, and a combination of indexed sequential and random. Further, it provides list file organization in two forms, a two-cell list, and a usual double-linked list (but not a multiple-linked list or ring structure).[4,14] As such it represents an improvement over the FORGE software which Burroughs has offered. FORTE is designed for use as an extension of COBOL, not as a free standing software package for file organization and use.

Another relatively new entry in the field is MARS from CDC. In giving the user the appearance of a range of file organizations, it like UL/1 relies primarily upon the strict sequential and indexed sequential file organizations. Like GIS noted earlier, MARS is a generalized system providing access, maintenance, and report capabilities. It does however provide the capability of building an inverted list organization. Its characteristics were still fluid at the time of preparing this paper.

### Non-vendor augmentation

The number of implementations of file organization alternatives are available in the software market from sources other than computer vendors. With IBM's Summer 1969 announced changes in software policy, this growth in alternatives can be expected to grow still larger. Only a brief selection is covered here, based primarily on age and popularity (see Table III).

Two distinct classes of offering are available in the software market. One uses and elaborates upon the vendor provided file organization and software support. Another replaces the vendor provided file organization and hence also provides its own software. A brief look at each of the groups will round out the comparison, since these offerings may soon become more popular in the market.

In the first group, some of the best known are the MARK-IV, the FILE EX, SCORE-II, and INQUIRE. The first two of these use the vendor-provided strict sequential and indexed sequential file organization techniques. To these they add an important software superstructure for report preparation, data retrieval, and file maintenance. As such they provide an alternative to the user for preparing his own programs to accomplish similar ends, and to the use of the vendor-provided software.

The SCORE-II also uses the vendor-supported sequential and an indexed sequential file organization. In addition it also provides tree structure, not directly but based upon a combination of the strict sequential and indexed sequential. This adds flexibility to the package of report preparation, retrieval, and maintenance facilities.

TABLE III—Summary of selected non-vendor augmentations

| *Strict Sequential* | *Indexed Sequential* | *Director Random* | *Other* |
|---|---|---|---|
| MARK-IV | MARK-IV | | DM-5 (hierarchy) |
| FILE EX | FILE EX | | SCORE-II (tree) |
| SCORE-II | SCORE-II | | INQUIRE (list) |

Differing in its choice of the underlying file organization is INQUIRE. This utilizes the indexed sequential and the direct or random file organizations. But these are not directly accessible to the programmer. Rather, INQUIRE combines them to form a modification of an inverted list file structure.* This gives added power to the file maintenance, retrieval, and report capabilities of INQUIRE. Access by attribute and by property is facilitated by the inverted list organization, but maintenance requires adjustment of the lists as additional operations.[4]

In the second group, the oldest and most publicized entry is the DM-5 (Data Manager-5) which has been described in the literature of the field.[8] DM-5 ,like the others, includes the software for retrieval, maintenance, and report preparation. DM-5 utilizes a hierarchical file organization of a compound form. Tables are used at several levels. Both random and next-record access is handled by use of the tables, and are of about equal speed for access by attribute. Since the records are not ordered by a key, but many keys can be used in the construction of the tables, the single key restriction of the positional file organization is avoided with a result similar to that for the inverted list file organization.

In summary, the non-vendor offerings in the software market typically combine into a single package both file organization and convenient aids to using it. The offerings thus far do not attempt to replace the file organizations supported by the computer vendors.

### COBOL extensions

The Data Base Task Group proposed last year to

* The inverted list was developed about 1964 under the leadership of Dr. Jack Minker as a modification of the inverted file. The inverted file organization was in use in the information retrieval field in the years 1957-1958. The inverted file is a positional file organization with an ordering determined by multiple keys. Records in the file reoccur as many times as they may have keys, which need not be the same from record to record. By contrast, an inverted list is a list file organization of a compound form. The main portion of the file need not be and usually is not in a list form. The key portion of the file is organized as a set of lists consisting of pointers for each key to records in the main file. Since as a practical matter, the links are unnecessary, common practice is to elide them. The result is conceptually equivalent to an inverted file with all records replaced by surrogates (a common practice now), and with the records drawn into a subfile of their own with no redundancy. (The inverted list can also be viewed as resulting from a consolidation of the links in one direction from a muble chain or multilist file.[4,12]) In net effect in their modern forms, and as a practical matter, an inverted list differs from an inverted file primarily in emphasis and manner of implementation.

the CODASYL COBOL Committee an extension of COBOL to incorporate provisions for the complex ring file organization.[6] Although the discussion devotes considerable attention to the other file organization techniques, the proposal is for the inclusion of only one of them, the complex ring. In substance, this is very similar to the IDS noted earlier. This discussion included with the proposal indicates that ring file organization can be used to simulate or serve as other file organizations, such as sequential, random, hierarchical or tree, and inverted file. Although not presented in the discussion, it can also be used as for muble chains or a multilist file organization.

One of the major objectives of the Data Base Task Group was to work toward keeping the description of data stored with the data itself. This is in effect an attempt to delay binding time. Since delayed binding time in general improves the flexibility and power of the resources available to the programmer, the objective is commendable. Providing linkage among data can be a definite step in this direction. The question to be argued is whether or not the ring file organization is the best choice of means for accomplishing this objective as well as serving as a worthwhile extension of COBOL.

From the comparisons presented, it can be argued that replacing a ring file organization by a frankly compound file organization sans links, would gain more for COBOL. Examples of candidate file organizations are the inverted list and the hierarchical. Access for both is faster and more powerful; maintenance for both is simpler.

## CONCLUSION

Automatic computers during the middle and late 1950's had by present day standards, relatively slow execution times and great restrictions upon the availability of both internal and external storage. The trend has been toward increasing the availability of larger and larger amounts of storage capacity, and toward faster and faster operating speeds.

These changing computer capabilities suggest the desirability of seriously rethinking the historic preference for positional organized files. This was certainly an appropriate choice of file organization technique, when storage capacity was extremely limited and operating speed was slow. It required the least storage space and the least direct overhead within the program at the time of file use. The positional organized file entails a very heavy cost of additional operating time in order to reorder (sort) the file. It also involves the time to rewrite the file periodically as a part of the maintenance of the file, depending for its extent upon the form of the positional file organization.

Now that computers have much more extensive external and internal storage capacity and operate more rapidly, it appears appropriate to reappraise our continued reliance upon positional file organization techniques. Let us consider briefly the alternatives. The attributed file organization is still too expensive of storage space and of machine time for serious attention in pure form. The list file organizations in general suffer from costly maintenance. The exception is the inverted list. The hierarchical file organizations appear attractive, but like the inverted list, are in practice compound file organizations.

It is significant that these latter two file organization techniques are generally not available to computer users because the supporting software is not generally available. The software exists, but the form of most puts it beyond the reach or scope of operations for most computer users. But this gap is narrower now than it was. Some vendors such as CDC and Burroughs have started to move to provide a wider range of file organization techniques. Independent software firms are starting to offer a wider variety of alternatives. But a gap still exists.

## REFERENCES

1 C W BACHMAN
*Integrated data store*
DMPA Quarterly Vol 1 No 2 Jan 1965 10-30

2 J H BRYANT P SEMPLE
*GIS and file management*
Proc 21st Natl ACM Conf 1966 Thompson Book Co Washington D C 97-107

3 R G CANNING
*Data management: file organization*
EDP Analyzer Vol 5 No 12 Dec 1967 14 pages

4 N CHAPIN
*A comparison of file organization techniques*
Proc 24th ACM Natl Conf 1969 ACM New York 273-283

5 N CHAPIN
*Data structures*
Automatic Computers N Y Van Nostrand Reinhold Co in press

6 Data Base Task Group
*COBOL extensions to handle data bases*
SIGPLAN Notices Vol 3 No 5 April 1968 1-45

7 M E D'IMPERIO
*Data structures and their representation in storage*
Annual Review of Automatic Programming Vol 5 Oxford 1969 Pergamon Press 1-75

8 P J DIXON S JEROME
*DM-1—a generalized data management system*
Proc SJCC Vol 30 1967 185-198

9 A J DOWKART et al
*A methodology for comparison of generalized data manage-*

*ment systems*
CFSTI No AD-811-682 March 1967 287 pages

10 IBM CORP
*Introduction to IBM System/360 direct access storage devices and organization methods*
IBM Corp 1968 White Plains N Y 70 pages

11 W I LANDAUER
*The balanced tree and its utilization in information retrieval*
IEEE Trans on Electronic Computers Vol 12 No 6 Dec 1963 863-871

12 D LEFKOVITZ
*File structures for on-time system*
Spartan Books 1969 Washington D C 215 pages

13 J MINKER J SABLE
*File organization and data management*
Annual Review of Information and Technology 1967 John Wiley and Sons Inc N Y 123-160

14 N S PRYWES H J GRAY
*Outline for a mutilist organized system*
ACM Natl Meeting 1959

# An information retrieval system based on superimposed coding*

*by* JOHN R. FILES and HARRY D. HUSKEY

*University of California*
Santa Cruz, California

The cost of storing information in machine-accessible form has declined markedly in the last decade, and promises are such that one can look forward to having complete libraries available in such form. This places increased importance on algorithms which make it possible to search large files efficiently.

This paper describes an approach to this problem. In practice, information in a large file can be more efficiently accessed if it is indexed in some manner. The method of indexing which will be discussed is particularly well suited for a file which:

1. Is very dynamic with both deletions and additions frequently occurring.
2. Contains an extensive vocabulary which is to be encoded.

Both of these characteristics are frequently found in files that are to be coded. A file of information on recently published articles about a given subject and a card catalogue for a large library are good examples of files which require a large amount of maintenance. If updating the index (code file) is expensive and time-consuming, updating is put off until it is felt that the performance of the system has deteriorated enough to justify the effort required to update it. Until the updating takes place, information which is no longer of use is still retrieved, and the new information, if present, is in a secondary file. Keeping a secondary file containing recent additions avoids the serious problem of not having new material available, but it does decrease the efficiency of the system since such a file must be searched separately each time an inquiry is made of the main file.

The ability to utilize an extensive vocabulary is also very important. In the proposed system the vocabulary to be used is derived directly from words used in the original documents, thereby eliminating the time-consuming and expensive practice of manually abstracting and choosing indexing terms. Machine-generated derivatives of the original vocabulary retain more information about the original content of the item than does the manual system of assigning descriptors. In the manual case when selected descriptors are assigned to a document, associations of descriptors to words and to phrases are made. Such associations are not made in exactly the same manner by two trained indexers, and it is likely that the associations made by the average interrogator of an information retrieval system will be even more diverse. Because of this lack of uniformity in assigning descriptors it is desirable to allow each searcher to ~~determine~~ words and phrases that he wishes to associate with the concept on which he is doing a search. Postponing such associations until the time of the search can be accomplished only if the entire word content is preserved in the coded form.

Ease of update and freedom of vocabulary are not enough in themselves to make a coding algorithm worthwhile. Factors such as speed of access, ability

* The research reported on here was done at the University of California at Santa Cruz with the partial support from Project Genie at the University of California at Berkeley (Contract SD-185 with The Advanced Research Projects Agency of the Department of Defense).

to make searches for combinations of words and compactness of code file are also important considerations. All of these characteristics will be discussed for the coding scheme outlined below.

*The system*

The information retrieval system which was investigated can be divided into three components: preparation of the text, generation of the code file, and the searching procedure. A general outline of the first two components can be seen in Figure 1.

Since the form and format of the text to be used can be expected to vary greatly, the text is standardized as it is read in. Flags are set to indicate boundaries between records as well as at the ends of lines to make it easier to reproduce the document when it is retrieved. Also, as a measure to reduce the bulk of the file generated (text file) extra blanks in the input text are removed. In the pilot system the text file was generated from two sources: a bibliography of computer science and a listing of authors and titles from recent issues of *The Computer Group News of the IEEE*. Both of these texts were read, processed, and stored on a disk. The text file generated was 100,000 characters stored one character per byte.

Once the text file is generated coding can proceed. The text file is examined character by character until the end of a string which is to be coded (word) is encountered. The unit coded is a string of at least three alphabetic characters surrounded by non-alphabetic symbols (an English word). After the word is found it is compared with a list of non-content words, (i.e, the Delete List containing words such as: of, the, and and). If the word is found in the Delete List there is no further processing of that word, and the next word is considered.

When a word is found that is not in the Delete List, the trimming algorithm is applied to reduce the word to a pseudo-root. Common endings such as s, ed, ing and compound endings such as fully (as in carefully) are removed. By removing endings, different forms of the same word are made into synonyms. For example, the words 'computer' and 'computers' will both be reduced to the base 'comput.' This derived root is then passed on to the coding procedure. (Further discussion of trimming algorithm in Appendix C.)

In the coding procedure, a code word is generated for each record. The code word can be thought of as a bit string containing N bits, all of which are initialized to zero at the beginning of the coding operation. When a trimmed word is to be coded into the code word, the numeric value of the letters in the word is summed,

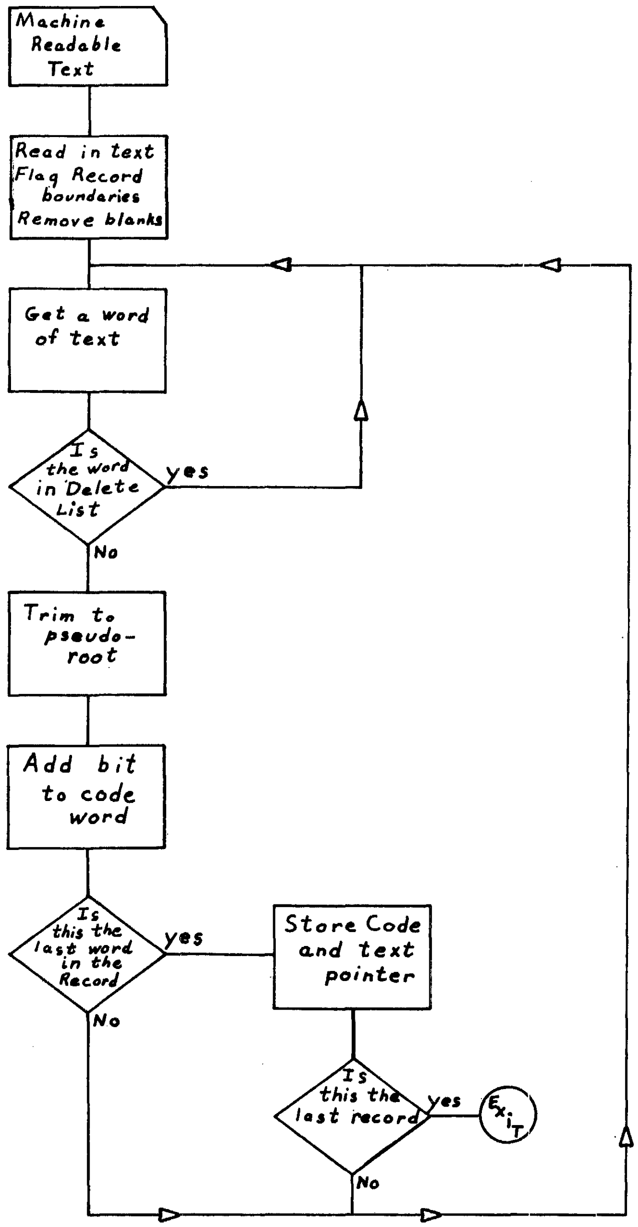
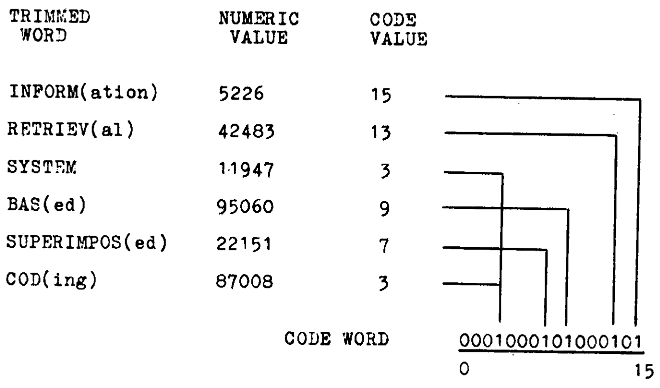

Machine Readable Text
Read in text Flag Record boundaries Remove blanks
Get a word of text
Is the word in Delete List
yes
No
Trim to pseudo-root
Add bit to code word
Is this the last word in the Record
yes
No
Store Code and text pointer
Is this the last record
yes
No
Exit

Figure 1—Coding procedure

giving a number which is used to choose an element from the uniform distribution of integers between 1 and N. Thus the resultant integer (code value of the word) is generated by an algorithm which given the same trimmed word in the future will generate the identical code value for that word. By using a fixed arithmetic procedure to produce the code value for a word, the need for a dictionary of words and assigned code values disappears. This frees the large amount of storage which such a dictionary would occupy as well as saving the time required to search such a file. If



| TRIMMED WORD | NUMERIC VALUE | CODE VALUE |
|---|---|---|
| INFORM(ation) | 5226 | 15 |
| RETRIEV(al) | 42483 | 13 |
| SYSTEM | 11947 | 3 |
| BAS(ed) | 95060 | 9 |
| SUPERIMPOS(ed) | 22151 | 7 |
| COD(ing) | 87008 | 3 |

Figure 0—Coding "An information retrieval system based on superimposed coding"

for a particular word the code value generated is K, then the K'th bit in the code word is set to one. (Figure 0).

The entire operation of finding a word, checking the Delete List to see if it should not be coded, trimming and coding is repeated until the entire record is processed. The code word which is uniquely determined by the words in the record is then stored in a file (code file) along with a pointer to the beginning of the record in the text file. This procedure is repeated until all the records have been coded.

After coding, the file is ready for searching. The searching program accepts any number of words, each of which is processed in the same manner as the words in the text file. It is looked for in the Delete List, trimmed, and used to generate a code value. This code value is then used to produce a query code in exactly the same way as the code words were produced in the code file. Upon generation of the query code the actual search may begin. Each code word in the code file is matched against the query code to see if the query code is a subset of it. (Here a bit string X is said to be a subset of another, Y, if when the I'th bit in X is one, the I'th bit in Y is also one, i.e., 1010 is a subset of 1011 while 0101 is not.) Each time that the query code is a subset of the code word, the pointer to the text file is used to gain access to the corresponding record which can be further processed to see not only if it contains the relevant words, but that the words are in the correct order.

The above is a brief description of the coding suggested for a file of an information scanning program. Some details such as the exact procedure for removing endings and the use of several independently generated code values to produce multiple code words for a given record, were not dealt with here. A more detailed treatment of these problems can be found in the appendix.

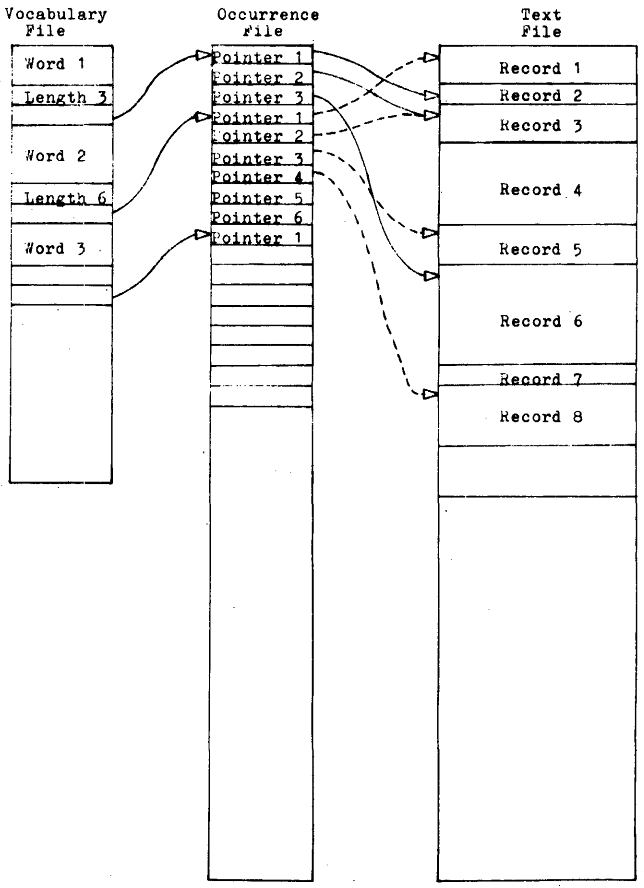


Figure 2—Inverted file

*Results*

From the pilot system, data was gained on the performance of such a system of superimposed coding. When possible, the performance of the superimposed coding system will be compared with that of a threaded list and inverted file. (Figures 2 and 3) The following factors received major consideration:

1. Ease of update
2. Effect of a large vocabulary
3. Amount and type of storage
4. Speed of search
5. Cost

Before making any comparisons it would be best to give a brief description of threaded lists and inverted files. An inverted file consists of two main parts, a vocabulary file and an occurrence file. As records are processed, each significant word is looked up in the vocabulary file. If the word has appeared before, it has associated with it a pointer to an area in the oc-

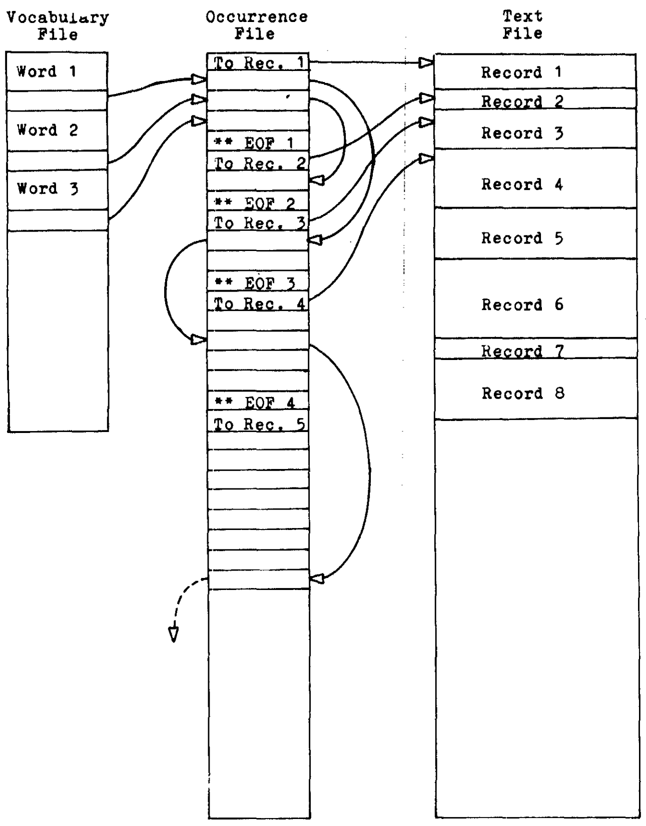


Figure 3—Threaded list

currence file; if not, then an area in the occurrence file is set aside for the word and a pointer to the first location in that area is entered in the vocabulary file. After this pointer is found, an entry is made in the first free location in the corresponding area of the occurrence file to indicate the record in which the word occurred.

The threaded list on the other hand, has the same type of vocabulary file, but the occurrence file is arranged in a different manner. The pointer in the vocabulary file now indicates a location associated with the first record containing the given word. This location in the occurrence file, in turn, contains a pointer to another location in the occurrence file associated with the second record which contains the word, and the pointer in this location points. . . Thus a linked list of all the occurrences of the word is formed.[2]

1. Ease of update

In the proposed system a record can be added or deleted very easily. To delete a record a search is performed which will retrieve the desired document. This produces not only the pointer to the record in the text file but the location of the record's code in the code file. The code word and pointer are removed from the code file, and their location is recorded as being free to be used for a new entry to the code file. The space that the text was occupying in the text file is now also free to contain new text. In order to add a record, which is the more common situation, the text of the new record is added to the text file in the first free location of a suitable size or at the end. It is then processed in the same manner as all the other records have been. The generated code word and pointer is inserted in the first free space in the code list. Here no room is wasted since all of the code word and pointer combinations are of the same length. Thus any type of update in the code file will affect only the code for the record which is being changed.

The threaded list can be updated with slightly more effort. The problem, and a minor one, is that the records in the occurrence file are not all of the same length, making it necessary to see if there is enough room in a given free area to insert the new entry.

The inverted file on the other hand is far more difficult to update than either of the others. If a record is to be removed all that need be done is to delete all pointers to it in the occurrence file. The addition of a record however becomes a serious problem. If for every word in the record there is room for an additional pointer in the areas set aside for pointers to records containing that word, then the update is easy. But if there is no room, a secondary file must be set up. The number of such files will grow until it is felt that a thorough update should be made. Then the entire text file must be re-inverted to produce a new vocabulary and occurrence file. This is a very time-consuming and expensive project.

2. Effect of a large vocabulary

With the superimposed coding there is no problem associated with having an arbitrarily large vocabulary. This is true because the superimposed coding does not require a table of vocabulary words like the inverted and threaded list files do. Since the vocabulary file is not present and does not have to be searched, increasing the vocabulary neither lengthens the time required for a search nor increases the amount of storage required to contain the coded information.

3. Storage requirements

The major advantage of superimposed coding lies in the great economy of storage. In the pilot program which was run, a text file of 100,000 bytes was used to produce a code file requiring 3,000 bytes. This reduction of 30 to 1 from the text to the code file is far better than the ratio obtained with the threaded list and inverted files. Such reductions are largest with small files such as the one experimented with, but substantial reductions do exist even in larger files. For example, assume that the text file consisted of 10,000,000 bibliographic entries, each containing 12 words which will be coded. Such an author-title entry was found to have roughly 300 characters in it, implying that the text file would be roughly $3 \times 10^9$ characters. Also assume that an average search contains at least three significant words. Such an assumption is made on the grounds that a search based on fewer words would tend to return more titles than would be of interest due to the very large size of the bibliography. From these two assumptions, utilizing considerations explained in Appendix B, it is found that the code file would consist of seven code words and one pointer for each record. Each of the code words is produced in a manner similar to the single code word mentioned before. Now, however, once the trimmed form of the word is found seven different procedures are applied to produce the pseudo-random number between 1 and N for each of the seven code words. Each of the code words will have 24 bits and the pointer will have 32 bits, thus indicating that each record will produce 25 bytes of code in the code file. The total size of the code file would then be $2.5 \times 10^8$ bytes, which still is a reduction of better than 10 to 1.

Such a reduction is far out of reach of an inverted file since each record in the text would have to have twelve 24 bit pointers pointing to it, and one 32 bit pointer from the record to the starting position of that record in the text file. This requires a total of $4 \times 10^8$ bytes and indicates only a portion of the room taken up by the inverted file. It does not include the vocabulary file which would be substantial, nor does it encompass the overhead of the occurrence file consisting of markers for the boundary between lists of pointers for a given word. Also it ignores the room which must be set aside for a linking pointer in case a new occurrence is to be added.

An additional advantage of the superimposed coding lies in the type of storage which can be used to store the code file. Since the file will be searched serially the storage media need not be random access. This permits the use of a cheaper sequential access storage device such as magnetic tape, which could greatly decrease the cost of such a system.

4. Speed of search

Evaluating the speed of a search using superimposed coding is difficult since the speed of any implemented system depends heavily on the characteristics of the storage media containing the code file as well as on the obvious consideration of the size of the text file. The search can be performed by reading the code file from bulk storage into addressable memory and comparison of the query codes with code words made by software. If this is done then the time required to search the code file can be cut to less than $6 \times$ (the memory cycle time of the machine) $\times$ (the total number of code words in the code file). This speed can be achieved due to the simplicity of the comparison which the software must make. The program only needs to test if X is a subtset of Y by loading the accumulator with Y, doing a logical AND of the accumulator with a register which contains X, and testing to see if the accumulator equals X. When large text files are used, and there are several independently assigned code words for each record, time is saved by being able to reject a record when any one of the query codes fails to be a subset of the corresponding code word. By taking advantage of this a substantial amount of time can be saved. In the previously mentioned large file, with seven code words for each record and an average search of three words, more than 90 percent of the records would be rejected after only the first comparison was made. This means that there would be 36 memory cycle times (the time allotted for the six comparisons which did not have to be made) free to take care of the overhead in the searching program.

Even with this simple and fast searching procedure, a search does require longer than the threaded list or inverted file. Although the implementation of this technique in software is slower, there are several methods that radically reduce the amount of time required to search the code file.

Since the algorithm for searching the code file is simple, the actual testing to see if X is a subset of Y can be done with very simple hardware. If the I'th bit of X is 1 and the I'th bit of Y is 0 for any of the values of I from 0 through 7, then X is not a subset of Y and the value of Z will be 1. If in no case is bit I of $Y=0$ and bit I of $X=1$, then X is a subset of Y and Z is 0.

Considering the speed of present day circuitry the time required to search a code file would be reduced to the time required to transfer the data from bulk storage. Since the hardware is so simple, it is practical to scan data from several sources simultaneously. An

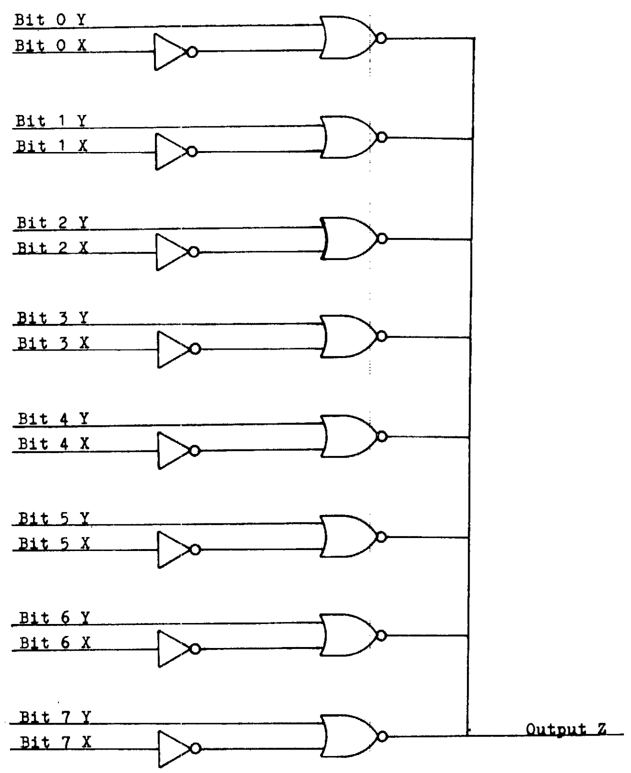


Figure 4—Hardware to test if X is a subset of Y
$Z = (\bar{Y}_0 \wedge X_0) \vee (\bar{Y}_1 \wedge X_1) \vee \ldots \vee (\bar{Y}_7 \wedge X_7)$

alternative to having the file searched externally would be to wire into read only memory the commands to test for a subset. By adding instructions to use the next code word and repeat the operation if the test fails, the search will proceed through core memory at a rapid rate making only one core access for each test. The end of the list of code words can be marked by a code word containing all ones. This has any possible query as a subset and would assure that the loop was interrupted at that point.

A second technique which would reduce the time required to search the file is to sort it in some manner. One such method which generates a superimposed 8 bit code from a 24 bit code is discussed in Appendix A. Other methods such as carefully dividing the code file into small groups and then doing a logical OR of the chosen code words to form rejector vectors have been suggested.[4]

In comparing the speed of the search it should be noted that with superimposed coding and when searching for several words, the search for all of the words is carried out at once. In the threaded list and inverted file a search for several words is made by making a list of occurrences for each word and then finding the intersection of the lists. Due to this parallelism of the search superimposed coding can handle a multiple word search in a more efficient manner than the other two methods.

At first glance it appeared that searching the entire code file would preclude the use of superimposed coding on a large file. With more careful examination, however, it is apparent that this type of code file can be searched as rapidly as either the threaded list or the faster inverted file.

Factors which lead to this conclusion include:

A. The code file search can easily be implemented in hardware. Such hardware is simple and very fast as well as being able to handle several streams of data simultaneously.
B. If several sequential access devices or a random access storage device is used then the code file may be structured to allow large blocks of the code file to be rejected with only one test.
C. The superimposed coded file is much more efficient at handling searches for records containing several desired keys.

## 5. Cost

The cost of implementing an information retrieval system utilizing the type of superimposed coding suggested would be substantially less than the cost of implementing a threaded list or inverted file using the same text file. The reasons for this stem from the reduced requirement for computational capability of the computer, as well as a substantial reduction in the amount of storage required for the coded information.

All three systems must dedicate a large amount of storage to the actual text. This, in all of the cases, can be either directly accessible to the computer such as a large disk file, or may be only machine referable such as a machine controllable microfilm display, like the proposed system at the University of California, Santa Cruz or the one being used as part of Project Intrex at M.I.T.[5] The difference of storage cost is not found in the storage of the text file but in the comparison of the cost of the storage of the code file of the superimposed coding system with the cost of storing the vocabulary and occurrence files of the threaded list and inverted file. The code file is smaller and can be stored in a sequential access device rather than a random access device. Both of these factors tend to reduce the cost of the system.

If scanning of the code file is implemented in hardware then the requirements on the computer become very small. All that it is responsible for is processing the words in the inquiry in order to generate the query codes, and then, while the search is in progress, stand by to store the pointers to the text file which the one or, possibly several, hardware scanners pass to it.

The trial program which processed the questions generated the query codes and handled the searching in software, was substantially under 16,000 bytes of code on an IBM 1130 with no overlaying. Thus the requirement for expensive core storage is low. The cost of the hardware which would do the testing for the query code being a subset of the code word and its interfacing with the computer would be very small compared to the cost of the necessary storage devices.

One phenomenon which is found in the superimposed coding and not in some other forms of coding is the presence of spurious matches. These occur because, in a given code word the fact that the I'th bit is zero signifies that any word assigned the code value I is not in the record. The converse is not true. Since many vocabulary words could cause the I'th bit to be one, the I'th bit being equal to one, does not indicate that a specific word is present. By generating several independent code words for each record the number of times that superimposing will cause an irrelevant record to be retrieved can be made arbitrarily small. Take for example the case where twelve words were coded into seven 24 bit code words. In that case the probability that a record in which all seven of the query codes for a question were a subset of the code words, and none of the three words involved in the search were in the given record, was $3 \times 10^{-10}$. (See formula in Appendix B, bd = .35, cw = 2.8, qc = 7)

Since the number of such spurious matches can be limited to any desired extent, although not entirely eliminated, it is convenient to perform some final verifying operation to assure that the words specified in the search are actually present. This verification in the case of the pilot program was accomplished as a side result of the check to see that the desired words occurred in the specified order. Consequently there was no penalty in making this extra check on the records which were retrieved.

The requirement that additional checking be done is not an unreasonable one. The fact that a document contains the words in which one is interested does not necessarily indicate that the document is of interest. Therefore any key word searching procedure can only be the first step of an information retrieval system. The job of a key word search is to quickly reject records that do not contain information of interest. In this sense any of the three types of key word information retrieval systems which have been mentioned are more properly information screening procedures which can rapidly eliminate a large portion of the text file as unlikely to contain relevant information. Such a system should be used to identify those records which warrant further and more extensive examination.

## CONCLUSION

The method of superimposed coding which has been discussed is a simple and relatively inexpensive manner of scanning a large text file. With a simple check for spurious matches made after the search, such a system can stand alone as a key word information retrieval system. On the other hand since the actual scanning of the text can be easily and rapidly handled by peripheral hardware, the method is very attractive as a first stage screening method. Although the prospect of having to search the entire code file for every inquiry, at first glance, appears discouraging, the simplicity of the scanning algorithm and the ease with which searches can be carried out in parallel makes such a linear search very reasonable.

## APPENDIX A

Besides implementation in hardware, measures can be taken to eliminate the need for searching the entire code file, thus reducing the required search time. One manner of doing this is to use the first code word of each record to generate a shortened code word for it. In the case of a 24 bit code word, the first bit of the of the second level code word is the logical OR of the first three bits of the first level code word. Bits 4 through 6 could also be ORed and used as the second bit of the second level code word. Continuing this process an 8 bit second level code word is produced based on the bits 1 through 24 of the original code word. Since there are only 256 of these second level codes possible, with each record's first code word being mapped into one and only one of these classes, the file is partitioned into 256 sets characterized by the numbers 0 through 255. When it is time to search the code file, the element of the partition that the first query code belongs to is determined. If for example the query code is 000100000010000001000000 it would belong to set 84 (01010100). The only sets which would have to be searched would be those characterized by numbers which have 84 as a subset (i.e., 11111111, 11111110, 11111100 would have to be searched, but 11111011 would not have to be examined further). There would be only 32 out of the 256 sets which would have to be searched, thus the number of code words which would have to be compared with the query codes would be reduced. Using the scheme of coding 12 words into 24 bits would cause roughly 10 percent of the code file to be classified as 255 (11111111) and just over 3 percent to be classified by a number whose binary representation contains 7 ones and one zero. Due to the non- uniform distribution of the code words over the 256 sets, the reduction in the amount of the code file to be searched would not be the 7/8 suggested by the reduction in the number of sets which must be searched. The reduction would, however, be in the neighborhood of 30 percent (3/8 of the sets whose binary representation has sevens one and one zero and 18/28 of those with six ones and two zeros can be eliminated).

## APPENDIX B

Since care was taken to assign the code values using numbers from a uniform distribution, the expected number of spurious matches can be predicted. By varying the length and number of the code words the frequency of spurious matches can be controlled. The number of spurious matches is a function of the bit density, bd (i.e., the number of ones in the code word divided by the number of bits in the code word); the number of code words per record, cw; the number of ones in the query code, qc; and the number of records which are coded into the code file N.

The expected number of spurious matches =
$Nx(bd)^{cw.q.c.}$

The number of bits used to code one record =
cw (the number of bits in the code word)

By keeping the number of bits used and the number of ones in a code word constant in the above two equations, it is found that the minimum number of spurious matches occurs when the number of bits in the code word is $e$ times the number of ones in the code word. That is when the bit density is $1/e$. The number of bits B to use for the code word when there are M words to be coded in each record is roughly 2.2M. This is found by considering that the probability that a given position will be left blank is $(1\text{–}1/B)M$. The expected bit density would then be $1\text{–}(1\text{–}1/BM)$. Setting this equal to the $1/e$ and solving for B yields the desired results.[3]

## APPENDIX C

The trimming program was divided into three sections. The first step removes all 'e's, 'd's and 's's from the end of the word. These letters were removed since there are many words such as 'attractions' which have compound endings terminating in s, es, d, and ed. By removing these letters, in the above, the suffix, 'tion', is left on the end of the word where it can be easily identified and removed in a later section of the program. Once this operation is completed the endings 'er' then 'ly' and then 'al' are searched for and removed if found. This procedure removes endings such as the 'ally' on the end of 'functionally' and again is a technique to handle compound endings.

After the above two trimmings have been accomplished, the Trim List is consulted. Suffixes found in the Trim List are arranged in order by length, starting with the longest. The ending found in the list is compared letter by letter with corresponding letters on the end of the word remaining after the first two trimming stages have been completed. Since all of the 's's, 'e's and 'd's have been removed, the suffixes are in an unusual form. For example, 'ness' would have been trimmed to 'n' by the first stage of the trimming procedure. Also 'ance' appears as 'anc' in the Trim List.

The reason for having suffixes in this form can be seen by considering the problem of trimming the two words 'finance' and 'financed'. In the second case, when the 'ed' is found on the end of the word, it is difficult to decide if the 'ed' or just the 'd' should be removed. The decision was made to remove the 'ed'. This means that to trim 'financed', 'anc' must be in the Trim List. However, 'finance' which should be reduced to the same pseudo-root requires either the ending 'ance' to appear in the list or the 'e' removed before the ending is compared with endings in the Trim List. The second course of action was chosen because it reduces the length of the Trim List and makes the first step of the trimming operation very simple.

The comparison of the endings in the Trim List is continued until either the list is exhausted or a match is found and the ending removed. There are two more checks to be made on the trimmed word. First, the last two letters of the word are compared. If they are the same, then the last letter is removed. This is
are the same, then the last letter is removed. This
done so that a word such as 'trimming' will be cut back to 'trim'. First the 'ing' is removed to give 'trimm, and then the second 'm' removed to give the desired root.

The final action provides some protection against trimming words too severely.The word 'deeds' would be trimmed to nothing. To prevent such loss of information, any word which has been reduced to less than three letters is restored to a length of three. At this point the word is considered trimmed.

There is one major problem which occurs with the

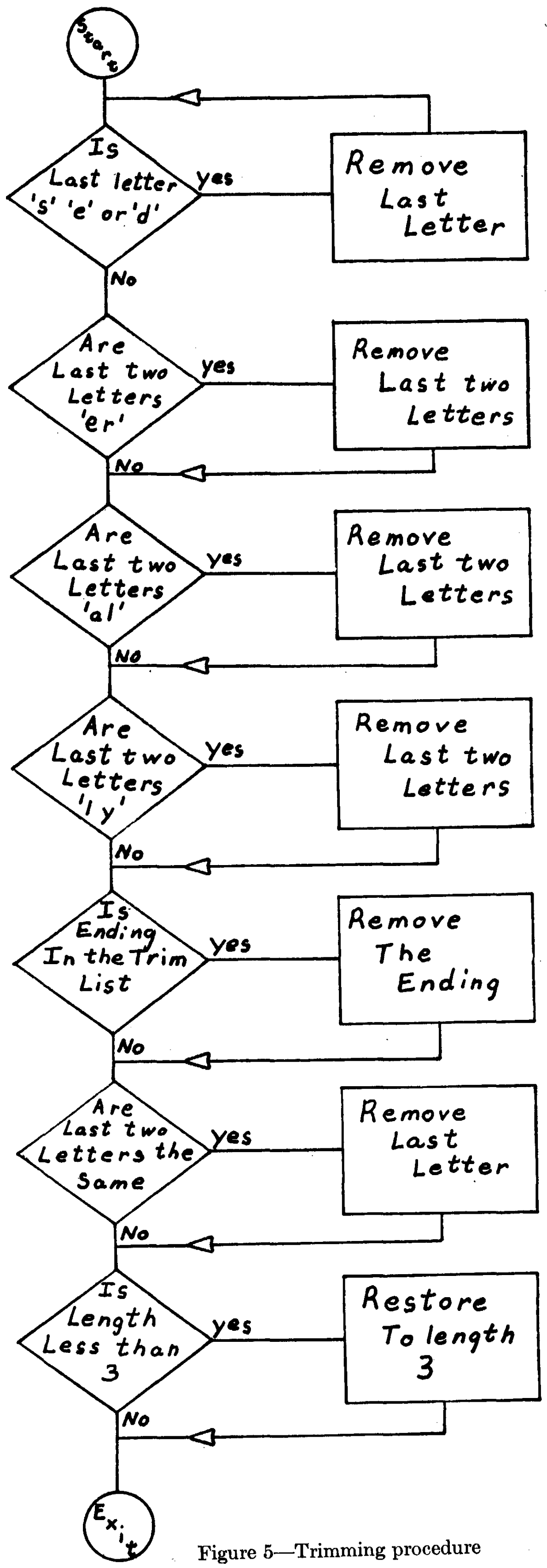


Figure 5—Trimming procedure

use of a trimming algorithm. Words which do not convey the same meaning can be reduced to the same root. An example would be that both 'information' and 'informal' are reduced to 'inform'. Such a result may be undesirable; it is unlikely that when searching for one of the words, the other would be of interest. Unfortunately the effect of this type of false retrieval could not be observed in the small pilot program. Such confusion of terms was rare due to the specialized nature of the text. In a system utilizing a larger text file containing a more generalized vocabulary, the number of such erroneous replies may become substantial. If a system utilizing a trimmed form of the vocabulary words is used for the first stage of an information retrieval system, the problem of such extra records is not a serious one, since the purpose of the search is to locate information-rich sections of the text. Further examination would determine whether the record is of interest or not.

The decision to utilize a trimming algorithm in the pliot program was based on the feeling that the error of failing to retrieve information was less tolerable than retrieving some irrelevant information.

TRIM LIST

| | | |
|---|---|---|
| ology | ful | i |
| ement | val | e |
| icant | ial | |
| ition | cal | |
| ation | ing | |
| orial | enc | |
| iting | anc | |
| ating | iz | |
| istic | ry | |
| ancy | iv | |
| ment | it | |
| ient | at | |
| ator | or | |
| ical | er | |
| ying | en | |
| ary | al | |
| eou | ag | |
| est | id | |
| ent | ic | |
| ion | ab | |
| ern | y | |
| dom | n | |

## APPENDIX D

### DELETE LIST

| | | |
|---|---|---|
| a | | |
| i | his | will |
| am | how | with |
| an | may | being |
| as | nor | would |
| go | our | every |
| in | the | might |
| is | was | other |
| it | also | since |
| so | does | their |
| to | from | there |
| we | have | these |
| all | more | which |
| and | must | while |
| are | that | would |
| but | this | should |
| can | thus | another |
| for | ways | however |
| had | were | either |
| | what | without |

## BIBLIOGRAPHY

1 R S CASEY et al C S WISE
*Mathematical analysis of coding systems*
Punched Cards Their Applications to Science and Industry
Reinhold PubCo 1958

2 T C LOWE
*Design principles for an on-line information retrieval system*
Doctoral dissertation submitted to the Univ of Pa 1966
Philadelphia

3 C N MOOERS
*Coding, information retrieval and the rapid selector*
American Documentation 4:225 Oct 1950

4 R T MOORE
*A screening method for large information retrieval systems*
Proc SJCC Vol 19 1961 259

5 C F OVERHAGE et al
*Massachusetts Institute of Technology*
MIT Project Intrex March 15 1968 to Sept 15 1968
Semi-annual Activity Rpt Cambridge 1968

6 E B PARKER
SPIRES (*Stanford Public Information REtrieval Service*)
1968 Annual Rpt to Nat Science Foundation
Project GN 600 742 Jan 1969

# Establishment and maintenance of a storage hierarchy for an on-line data base under TSS/360

*by* JAMES P. CONSIDINE and ALLAN H. WEIS

*Thomas J. Watson Research Center, IBM corporation*
Yorktown Heights, New York

## INTRODUCTION

As on-line interactive systems increase in popularity, several problem areas become more and more apparent. One of these is the management of the on-line accessible data base. It has been the experience of installations throughout the country that such a data base tends, if ungoverned, to increase in size as the system continues in operation, bounded only by the size of the storage available to contain it. It is, therefore, essential for the continuance of a viable system that this data base be examined and methods devised to control its growth.

In the first section of this paper we record some observations we have made on the nature of one particular on-line data base, specifically its growth and usage characteristics. The second section details a system we have designed to control the growth of the data base and insure maximum utilization of the on-line devices available. The third section describes the results of operating with the system. The fourth section details future amplifications and modifications to overcome some foreseeable difficulties in the present version. Finally we summarize our observations and re-state the conclusions we have reached.

### *TSS/360 data base at T. J. Watson Research Center*

Since our system first went on a somewhat regular schedule of four-hour-a-day user sessions in June 1968, it was clear that, even under these conditions of relatively low availability, managing the on-line storage was going to be one of our primary problems. The amount of on-line storage occupied by user data sets at that time was approximately 20,000 pages, or 80,000,000 characters (1 page = 4096 characters or 8192 hexadecimal digits). It was a matter of a few months before the amount rose to what is our working optimum, 30,000 pages or 120,000,000 characters. This optimum is dictated by the maximum number of devices we wish to devote to on-line storage. The distinction between devices and volumes should be made clear. A volume is a unit on which data are actually recorded. There are in principle large numbers of volumes available. A device is a unit on which a volume is mounted and which carries out the transmission of data to and from the volume. Devices are necessarily limited in number. A tape reel is a volume; the tape drive is a device.

To return to the data base, observations made at the time indicated that perhaps 10-20 percent of this data was non-useful. Examples of this are data sets defined but not used and never erased, output listings of assemblies and compilations done many days previous to the current date and other such system- and user-generated residues. Measures were devised to periodically and systematically remove such unwanted data from the on-line storage, thereby achieving a small amount of leeway while the problem was being further studied.

In an effort to acquire information on the usage of the data base, we implemented a means of marking a data set with the date on which it was used. Report programs were written to process the data thus recorded and the results were very informative to management and system programmers alike.

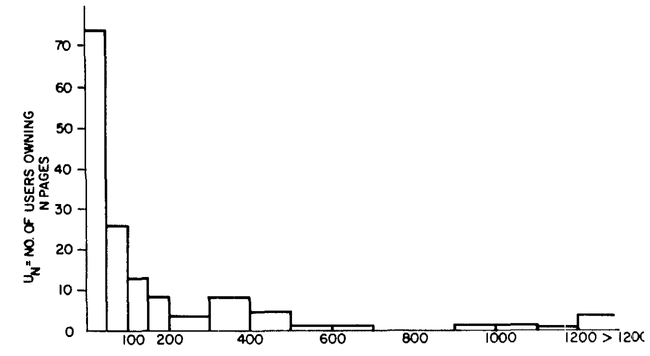
Extracts from a typical report are presented in Figure 1. Among the facts which can be determined from such reports are the names of the authorized users actually using the system currently, how much storage each user is occupying, how much he is using, and how the amount of storage used by each user varies from observation period to observation period. The total amount of on-line storage that is being currently used by all users is also recorded. In addition, the data recorded can be processed to yield an on-line storage profile, as shown in Figure 2.

For instance, in the reports formulated from data gathered on February 1, 1969 we discovered that of our 160 or so authorized users, some 50 had actually used the system since the beginning of the year. We also found that most of these 50 were not actually using all of the storage they were occupying. In one case, up to 95 percent of the storage of a particular user had not been used during the period. In total we discovered that of some 28,000 pages of storage on the system only 13,000 pages had been used in the last month. These figures were based on information recorded after all the waste space occupied by obviously useless data had been reclaimed. Similar data are being recorded periodically to monitor in a limited way the interactions of the users with the system. The amount of available on-line storage is recorded every time the system is loaded into the machine, a process which takes place three or four times in a fourteen hour day. The usage characteristics are recorded much less frequently, perhaps once or twice a month. Thus far, observations on this somewhat expanded time scale have been more than sufficient to give evidence of imminent difficulties in the matter of on-line storage.

Even though these measurements were made under conditions of limited availability, they gave clear indication of the existing problems involving the management of the on-line data base and the control of its size. We realized at an early stage that unless some steps were taken to reduce the amount of data main-

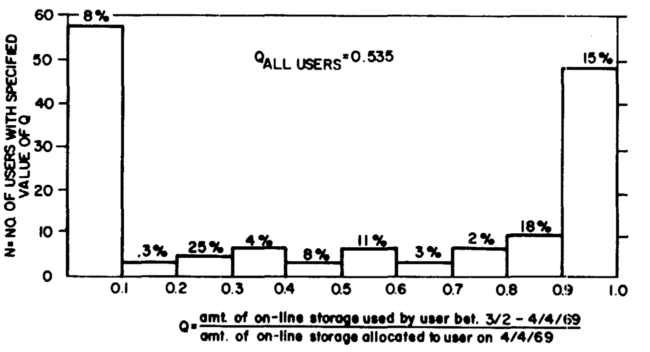
```
Date data recorded  = 4/4/69

Date for comparison = 3/2/69


Total Pages Used Since 3/2/69    =  14441

Total Pages In System on 4/4/69  =  27662




   USERID      PAGES USED        TOTAL PAGES

   USER01         1588               1751

   USER02           11                 18

   USER03            0                  7

   USER04          263                431

     .               .                  .

     .               .                  .
```

Figure 1—On-line storage reports



NOTE: The 4 users who own more than 1200 pages each own about 40% of the available on-line storage

Figure 2a—On-line storage ownership profile



NOTE: The number at the top of each column is the fraction of on-line storage owned by users in that category

Figure 2b—On-line storage usage profile

tained on-line, it would be impossible to operate the system in our user environment with the on-line storage capacity then available. As indicated earlier, the problem is by no means unique to our installation. Various means have been adopted to handle the problem of controlling the size of on-line data bases. One installation requires that each user validate every file he wishes to retain once in every twenty-four hour period. Unvalidated files are erased. Another approach, similar in some respects to the one which we will describe, is the "Date Deletion" scheme which has been in effect for some time on the Compatible Time Sharing System at Massachusetts Institute of Technology.[1]

Since we felt at that time that we did not want to place the primary burden of storage management on the user, we looked for some systematic way of restricting the amount of data stored on-line. We wanted to combine ease of operation with convenience for the users. It seemed clear that a potentially vast condensation of the on-line data base could be achieved by systematically moving unused data sets from on-line storage to demountable storage volumes. The underlying assumption would be that the overhead involved in restoring data sets that might be required by the users would be small compared to the advantages to be gained by being able to reduce the amount of on-line storage required at any one time. There were no observations of actual data set usage available to verify such an assumption, or to support any alternative, so we proceeded to implement a simple design to alleviate in part our pressing problem, and also at the same time to provide the experience necessary to evaluate the underlying assumptions. This "data migration" scheme is described in the following section.

### *Management of the on-line data base*

Because of the limited amount of on-line storage available, it appeared necessary to us to establish a hierarchy of storage volumes, ranging from high-speed permanently mounted direct access volumes to low-speed demountable magnetic tapes. The establishment of such a hierarchy immediately implies a mechanism for distributing data among the various classes of volumes according to some predefined or even dynamically defined criteria.

Initially, in TSS/360 three categories of storage volumes suggest themselves: first, on-line direct-access volumes; second, off-line direct-access volumes, which would require mounting to enable the retrieval of information from them; and third, tape volumes, which would require mounting and, of course, have a lower data transmission rate than the direct access volumes. The first category comprises what are described in the TSS/360 system literature[2] as public storage volumes. Categories two and three are handled by TSS as private storage volumes. In our discussions below, the term "archival" storage will be used to refer to storage volumes of categories two and three which are processed by the migration scheme. As far as the rest of TSS/360 is concerned, these volumes constitute a subset of the general class "private storage volumes".

The criteria to be used to govern the arrangement of data among the categories of volume are obviously the subject of wide differences of opinion. We have been limited in our considerations of this topic by the information that can be collected on our system about the usage of individual data sets. We have chosen to base ourc riterion on the information mentioned in the first section, i.e., the date on which the data set was last used. Specifically, a data set is useful or not depending only on the length of time since its last use. This is admittedly a very simple basis for judgment but for the moment it is what is available. Alternatives will be discussed briefly in the fourth section. The scheme has been designed to enable easy inclusion of other migration criteria as they are deemed necessary and the required information becomes available. It has been implemented in the form of seven commands and an auxiliary data set, which records the status of the data sets moved to archival storage. The commands are RMPS, MPS, EMDS, LMDS, RMDS, SAVE, and CMS. The data set is called SYSMDS. A brief description of each of these commands and the data set follows:

### RMPS—Recreate and Migrate Public Storage

This command and the one that follows, MPS, are modifications of the TSS/360 system command, RPS (Recreate Public Storage).[3] The RPS command is used to copy the contents of current public storage, one volume at a time, onto a new set of public volumes, leaving behind in the process useless data sets and producing a new system with cleaner public storage. The RMPS command adds the criterion of currency to the criteria of usefulness already in the RPS command. If a data set fails this test of usefulness, instead of being copied onto the new public storage it is copied or "migrated" onto an archival volume and cataloged. In addition, relevant data regarding this "migration" are recorded in a special data set called SYSMDS. The format of this data set will be discussed later.

The fact that the data sets which have been moved to archival storage are cataloged requires some elucidation. Having these data sets cataloged is important so as to prevent the duplication of data set names on public storage and on archival storage. Only one entry for a given data set name may appear in the catalog for each user. Private volume handling, however, is a sensitive area of TSS/360. If a user requests that a private volume be mounted and his request is granted, the device on which the volume is mounted remains assigned to him until he specifically releases it or terminates his session. Thus, if the user were allowed to directly access an archival volume simply by requesting one of his migrated data sets from the catalog, this volume could well remain mounted on the device for several hours. This would render it almost indistinguishable from a public volume, and defeat the purpose of the migration scheme.

We have avoided this by specifying the first three characters of the volume identification of all our archival storage volumes as 'SAV'. A minor modification to the system prohibits the user from directly accessing any volume whose identification begins with these three letters. Commands described below perform any service he may require which involves these volumes and always release the volumes, thus freeing the device, as soon as the service has been performed. This assures that devices will be in use as little as possible for purposes dealing with the handling of archival storage

## SYSMDS—Migrated Data Set Record

It is appropriate at this point to discuss in some detail the SYSMDS data set. It is an indexed sequential data set with an entry for each migrated data set with the data set name as key .These entries have the format illustrated in Figure 3. As well as being a record of the migration, the information stored in SYSMDS is also sufficient to recreate the catalog entries for the data sets moved to archival storage. In addition, there is an entry for each of the archival storage volumes. Included in these entries are the amount of available space on each volume, the number of pages to be erased, the number of erase pending data sets, and the total number of data sets on the volume. These entries have as key the nine characters 'ZZZZZZZZ.' followed by the six character volume identification. There is also a record containing the total number of archival pages erased and the total number restored to on-line storage. The information is all in EBCDIC characters so as to make it available by simply printing SYSMDS. An up-to-date copy of the SYSMDS data set is made after each modification and stored on the system residence volume (to insure

| LOCATION | CONTENTS |
|---|---|
| 0-44 | Data Set Name |
| 49-50 | Data Set Organization (Sequential,Partitioned,etc.) |
| 53-56 | Number of Pages(Size of the Data Set) |
| 59-64 | Date Created -'DDD/YY' (010/69 indicates the tenth day of 1969) |
| 67-72 | Date Last Used |
| 75-80 | Date Migrated |
| 83-88 | Archival Volume Identification |
| 91-94 | Archival Volume Type |
| 97-99 | File Sequence Number(for tape volumes only) |
| 102 | 'Erase Pending' (has the value 'Y' for Yes and 'N' for No) |

NOTE :

For a full discussion of TSS/360 terminology please consul Reference 2.

Figure 3—Format of the data set entry in SYSMDS

continuity between successive versions of public storage).

## MPS—Migrate Public Storage

This command differs from RMPS in that when it operates on current public storage it moves only those data sets which fail the test of currency. They are moved to the appropriate archival storage volume and the copies in public storage are erased. Appropriate entries are made in the SYSMDS data set. This command can also be applied to archival direct access storage volumes, producing an additional level of storage on tape, creating the three-level structure described earlier. Again, the entries in SYSMDS are amended to reflect the changes brought about by the execution of the command.

The two commands RMPS and MPS are the primary means by which out-of-date files are moved from noline to archival storage. The next group of commands is concerned with enabling the user to examine the contents of archival storage and modify the number and status of his files which are stored there.

## LMDS—List Migrated Data Sets

This command enables the user to determine which of his data sets have been moved to archival storage.

In addition to the name of each data set, information such as its organization, size, date last used, date migrated, etc., is provided.

### EMDS—Erase Migrated Data Set

This command enables the user to specify that a data set of his which is on archival storage is to be erased. This command simply marks the appropriate entry in the SYSMDS data set as "erase pending" for subsequent processing by the CMS command (q.v.). The data set name is not removed from the catalog until the actual erasure on the volume has been carried out by the CMS command. The user may specify that either a specific data set or all his data sets on archival storage are to be erased.

### RMDS—Restore Migrated Data Set

This command enables the user to bring about the return of a data set from archival storage to on-line public storage. The process occurs while the user waits. The data set is copied from the appropriate archival storage volume onto on-line public storage and the copy on archival storage is erased. Appropriate entries in the SYSMDS data set are amended to reflect the results of this operation. The archival storage volume is then released, making the device again available for allocation.

### SAVE—Put A Copy Onto Secondary Storage

This command enables the user to specify a data set as one to be migrated at the next execution of the RMPS or MPS command.

The maintenance of archival storage is carried out by the use of two commands. The first, MPS, discussed above, can be applied to the demountable direct-access 'SAV' volumes to produce a second level of archival storage consisting of data sets whose last use is more remote in time than those on the first, direct-access level. These would generally be stored on tape. The second maintenance command is the CMS command which will now be described.

### CMS—Clean Migrated Storage

This command examines data set entries in the SYSMDS data set for the "erase pending" flag set by EMDS to indicate that the corresponding data set is to be erased. The data sets are erased if they are on direct access volumes. In any case, the entries for the data sets in SYSMDS are deleted and the appropriate volume entries are amended to reflect the results of these transactions. If the number of valid data sets on a tape volume becomes zero, the tape is released or made available for further use for migration.

*Results observed after migration*

The first migration was carried out on March 10, 1969 in the process of converting our system from Version 2.0 to Version 4.0 of TSS. The criterion used was that a data set should have been used since January 1, 1969 to remain in on-line public storage. Operating problems prevented the processing of two of our six public volumes at that time. In the ensuing month an additional 3,000 pages were moved to archival storage. It should be pointed out that if the amount of data which was moved to archival storage had been returned completely to the current on-line public storage, we would not have had enough devices available to contain it. Thus the project did not simply justify itself; it proved essential to the continued life of the system.

Since that time the process has been carried out at approximately one-month intervals. The status of on-line storage as of July 1, 1969 is reflected in Figure 4 which is presented for comparison with Figure 2. It can be seen from Figure 4 that the overall characteristics of the on-line data base have not changed a great deal in the intervening three months. There are about twenty more users owning data sets on-line than there were in April, but the ownership profile remains almost exactly the same. Figure 4b reveals a noticeable increase in the degree of utilization of on-line storage. This is indicated on the whole by the increase in the value of Q, the utilization quotient, calculated for all users, and in detail, by the shift toward higher values of Q, especially visible between $Q=0.7$ and $Q=1.0$. Figure 5 contains similar information for the total storage on the system, i.e., on-line storage plus archival storage. This total storage is what would have to be stored on-line in the absence of migration, assuming there were enough devices to do so. The total storage, thus defined, is about 51,000 pages, of which some 32,000 are on-line and about 19,000 are archival. One can observe that the shape of the total storage ownership profile (Figure 5a) is very similar to that of the on-line storage profile. Figure 6 gives an idea of how the amount of storage occupied is divided between archival and on-line. Looking at this figure, one should be aware that there are thirty-six users who have *no* on-line storage, and thus cannot be classified as active. They are taking advantage, consciously or unconsciously, of the archival property of the migration volumes and leaving all their data stored in this fashion It should be pointed out that we have

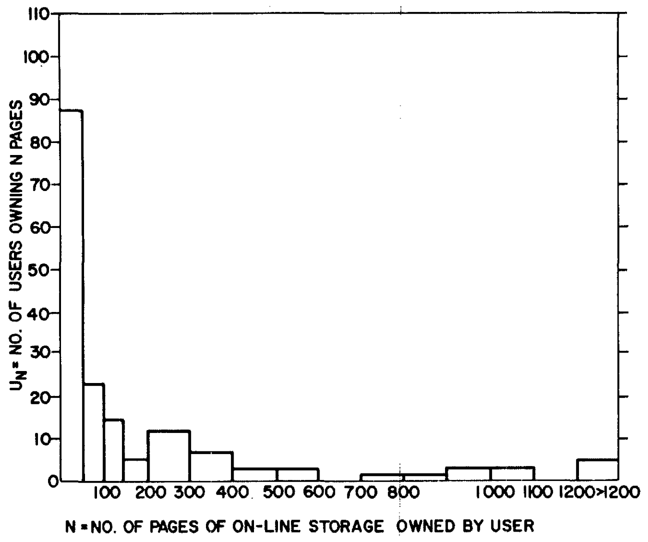


Figure 4a—On-line storage ownership profile on July 1, 1969

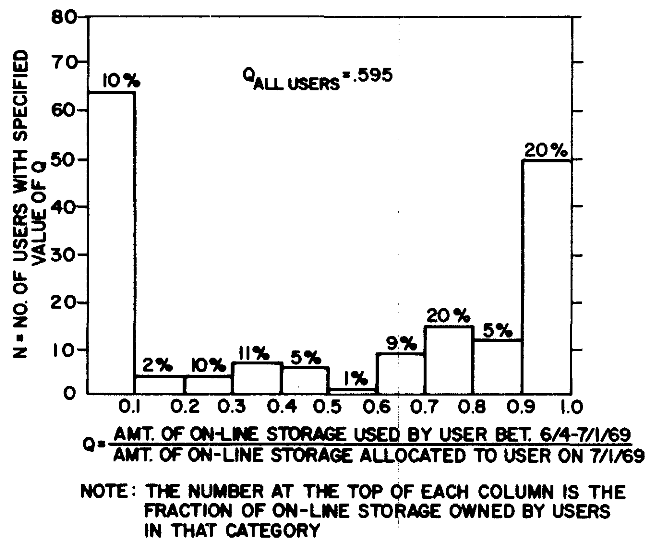


Figure 4b—On-line storage usage profile on July 1, 1969

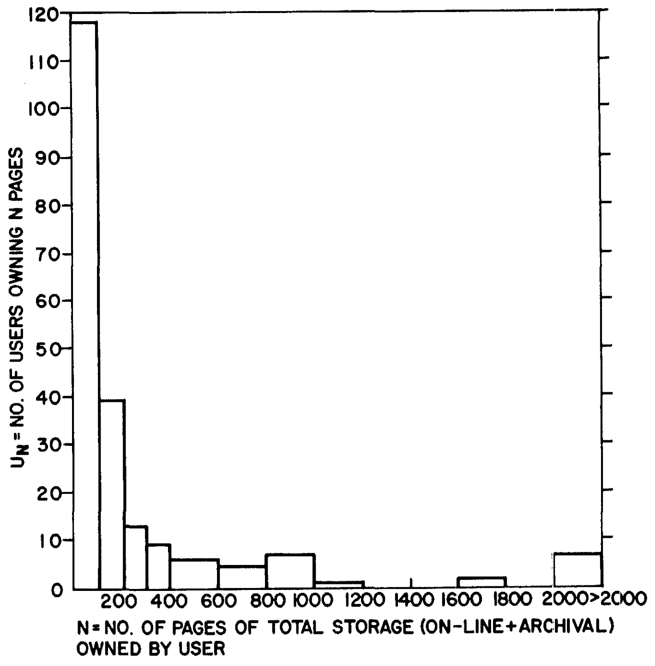


Figure 5a—Total storage ownership profile on July 1, 1969

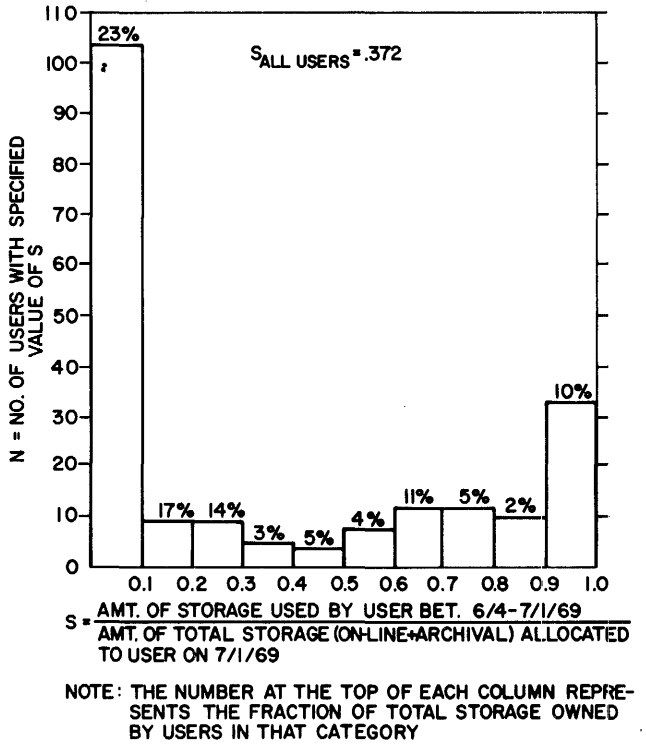


Figure 5b—Total storage usage profile on July 1, 1969

made no effort to encourage our users to police themselves in their use of on-line storage. Thus these figures must not be considered as reflecting what storage space the users need, but rather what they will occupy and use if they find it available. An accounting procedure is being instituted which may result in reductions by the users of the amount of on-line storage they occupy. This approach has been used with success in other applications, e.g., at Stanford University.[4]

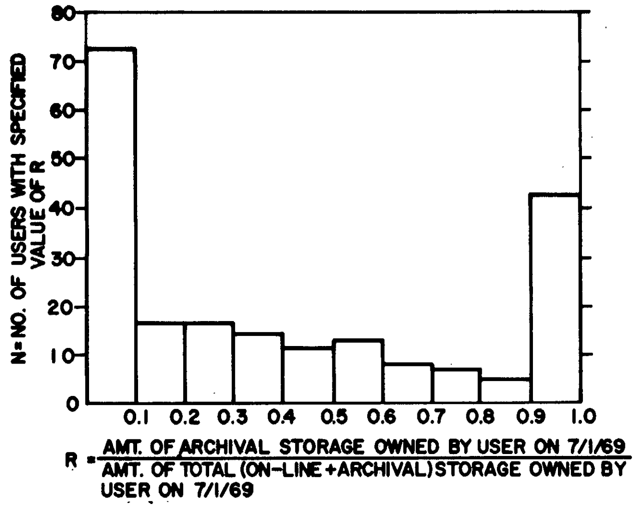


Figure 6—Archival storage/total storage ratio Distribution

| PAGES ON-LINE | PAGES MIGRATED | PAGES RESTORED | PAGES ERASED |
|---|---|---|---|
| 32,000 | 19,400 | 900 | 1100 |

Figure 7—Status of storage as of July 1, 1969

The status of migrated storage as of July 1, 1969 is presented in Figure 7. The small fraction of the migrated storage that has been restored to on-line storage is a favorable sign for the continued success of the approach.

Not at all surprisingly, in our experience with the operation of the migration scheme, several drawbacks have become apparent. For instance, one of the more valuable features of TSS/360 is that it allows users to share files with one another. This is made possible by links established in the system catalog between the directories of the individual users sharing the files. Under the present migration scheme, it is not possible for these links to survive the migration or restoration process. Thus after a migrated data set has been restored to on-line storage, the users sharing it have to re-establish the linkages which make the sharing possible. Another shortcoming from the user's point of view is that he is made aware of the existence of migration whenever he attempts to re-activate a file that has not been used recently. A separate action is required to make his file available to him once more. Also the criterion for migration is too simple to satisfy either the system manager or the user. For the manager, it is too easily circumvented, while for the user, it does not sufficiently distinguish between the user who occupies a large amount of on-line storage and the user who has a much smaller amount allotted to him. When migration takes place either one may find that his data sets have been migrated, and in fact the smaller user may find that more of his data have been moved to archival storage than the large user's.

*Amplifications and extensions-the evolution of migration*

There are several areas in which improvements are projected. These might be stated as goals in the implementation of a good migration scheme.

a. Migration should be transparent to the user except for the wait involved while a data set is restored to on-line storage. No action of the user other than his wish to use his data set should be required to activate the restoration process.
b. There should be reasonable criteria for migration and the information necessary to evaluate them should be available.
c. There should be a migration 'monitor' to determine the extent to which migration is necessary based on the condition of public storage, the amount of storage available, etc. In addition, based on system load, the monitor would schedule the migration process so as to have a minimum impact on system performance.

We are attempting to address a. and c. in a unified way. The first step is, of course, to allow the migration routines to be invoked by other programs as well as by commands from the terminal. Then the transparency problem can be handled by having the routines which supervise the user's access to his on-line data sets recognize that a data set has been migrated and initiate the process of restoration of the data set to online storage. The next step will be having the migration to archival storage activated by a routine which from time to time monitors the state of on-line storage and determines when more on-line space is required. Thus the necessity of programmer or operator intervention to initiate the migration process will be eliminated.

In a parallel effort additional information on the usage of data sets and on-line storage will be accumulated. As a simple example, we intend to add to the 'date last used' which we now record on the data set, information about the frequency of use of the data set. We hope then to be able to form reasonable judgments about which data sets to select for migration on the basis of this additional information. We also expect to take advantage of accounting routines to acquire information about the users and their use of the system. By accumulating as much information as possible we will be able to formulate more and more reliable criteria for the usefulness and currency of on-line data sets.

## SUMMARY

In summary, we have seen that the size of the TSS/360 on-line data base increases rapidly with use of the system. Since a limited amount of on-line storage is available, it is necessary to control this growth. Observing that at any time much on-line information is not being used, we have formulated a systematic method of allocating data sets to on-line or archival storage based on some criteria of usefulness. The elementary scheme put into operation at our installation has proven of great value in containing the on-line data base while giving the users an environment in which to expand their applications and use of the system.

We have come to several conclusions regarding the maintenance of our on-line data base which we restate here.

1. Some means of controlling the size of the on-line data base is absolutely essential for the continued operation of the system in our environment with our limited amount of on-line storage.
2. On the basis of our experience thus far, it is sufficient to examine the usage of data sets on a weekly basis or even less frequently to keep our on-line data base of manageable size. We do our cleaning up operations at approximately two-week intervals, with migration being carried out when necessary to reduce the size of the on-line data base to the desired value.
3. It appears that the amount of space gained by moving less used data sets to archival storage more than repays the effort involved. Most of the data moved to archival storage have stayed there. This is in part an indication that the criterion we have used for migration is a reasonable one, at least for our installation.

We intend to expand this scheme to make it as unobtrusive as possible while still continuing its work of limiting the size of our on-line data base. In addition we will continue accumulating information on the characteristics of our users and their interactions with the system so as to formulate the most significant criteria possible for migration.

## ACKNOWLEDGMENTS

The authors would like to gratefully acknowledge stimulating discussions with Ronald M. Rutledge and Albin L. Vareha of Carnegie Mellon University and Lee Varian of Princeton University. We are also indebted to Barry Hazlett of IBM Pittsburgh for the implementation of the CMS command.

## REFERENCES

1 MIT Computation Center
*The compatible time-sharing system: A programmer's guide*
MIT Press 1963 26-29 Cambridge Mass
2 W T COMFORT
*A computing system design for user service*
Proc FJCC 1965 Spartan Books Wash D C
C T GIBSON
*Time-sharing in the IBM system/360 Model 67*
Proc SJCC 1966 Spartan Books Wash D C
A S LETT W L KONIGSFORD
*TSS/360: A time-shared operating system*
Proc FJCC 1968 Thompson Book Co Wash D C
*TSS/360 concepts andfacilities*
IBM Document C28-2003 IBM Corp 1968
3 *TSS/360 system programmer's guide*
IBM Document C28-2008 IBM Corp 1968
4 N NIELSEN
*Flexible pricing: An approach to the allocation of computer resources*
Proc FJCC 1968 Thompson Book Co Wash D C
5 R C DALEY P G NEUMANN
*A general purpose file system for secondary storage*
Proc FJCC 1965 Spartan Books Wash D C

# Resources management subsystem for a large corporate information system

*by* HO-NIEN LIU, WILLIAM S. PECK
and PAUL T. POLLARD

*Pacific Gas and Electric Co.*
San Francisco, California

## INTRODUCTION

In the past quarter century, from MARK 1 (1944) ENIAC (1946) to IBM-360/195 and CDC-7600, the information processing community has progressed in diametrically opposite directions. On the one hand, the hardware[1] and software[2] development has been toward a general purpose computer system. On the other hand, the computer users often dedicate a general purpose computer for a special application where only parts of the computer system resources are used.

There are rarely any special computer applications which will utilize a general purpose computer to its full capacity in a balanced fashion. A corporate information system with its comprehensive application spectrum[3] will exploit the full potential of a general purpose computer system.

To assure a successful marriage between the comprehensive corporate information system and a general purpose computer system one must resolve the probelm of how to direct the different application subsystems to the desirable computer resources. After this problem is resolved, all the application subsystems within the corporate information system will operate in a homogeneous environment to obtain the optimum efficiency of the system.

The rest of the paper will describe a resource management subsystem designed and implemented with the above objective in mind.

### *System requirements*

The following requirements are essential for an effective resources management subsystem for a large corporate information system.

1. *Provide for the Orderly Execution of Programs*

   All applications and systems programs must function in harmony within a large corporate information system to ensure reliable and efficient operation. For instance:

   - Coordinating the execution of various asynchronous subsystems:
     - File/data management subsystem
     - Teleprocessing subsystem
     - Systems service facilities
     - Various applications subsystems
   - Prevent contention of usage of hardware and software facilities to provide optimal use of these resources.

2. *Support of a Variety of Applications Subsystems*

   A comprehensive corporate information system must contain many operationally independent yet logically interrelated application subsystems. The resources management subsystem should be able to support applications in a manner best suited to the needs for each individual project without placing undue restrictions on the others.
   Following are a few representative types of applications.

- Real-time, high volume, random arrival transactions require large numbers of programs to process them. For example, a simple inquiry response application which services the general public.
- Real-time, low volume random arrival transactions require a small number of rather complex programs to handle them. For example, on-line computation and update of the data base requires more processing and safeguard considerations than does the inquiry type of transaction.
- Batch, extra high volume sequential inputs require extraordinary complex processing logic. For example, maintaining a master file with several million large records involves the insertion, deletion, and update of files as well as the detail analysis of the data. It also includes the generation of scores of reports and intermediate files as input to other subsystems.
- Batch, medium or low volume processing uses time-consuming multiple file searching strategy to produce summary reports. For example, as the result of exception conditions many interrelated data are analyzed, summarized, and reported to aid management's decision making.

3. *Operate in a Multi-Level, Multi-Programming Mode*

Because of high volume, real-time applications which process a large number of messages, conventional multi-programming techniques cannot keep abreast of the traffic. Therefore, it is necessary that the system be operated in a multi-level multi-programming, or subtasking environment within a single region of main memory.

For flexibility in operation, the system should also have the following capabilities:

- Ability to suspend a subtask in one region without endangering the operation of the other subtasks in the same region.
- To transfer a subtask out of a region in order to be able to schedule a different subtask in its place.
- To change priority dynamically to provide the best system throughput.

4. *Simplicity of Operation*

When operating a real time system with a large network of remote terminals, events occur far more rapidly than the best computer operator can respond to them. Therefore, the resources management subsystem should have facilities to perform the following functions:

- Minimize the job control data required.
- Minimize operator decision and intervention.

5. *Test Mode of Operation*

In order to properly test programs to be placed in such a complex environment, a test mode of operation should be provided to perform the following functions:

- Allow program testing in an operational environment.
- Assure data security for those files which are accessed from or directed to the corporate data base.
- Assure that no actual updating of any of the corporate data base files occurs either purposely or accidentally.

6. *Statistic Acquisition and Reporting*

A continuous evaluation of the system should be maintained on a day-to-day basis. Statistics can be gathered during the processing day and reported at the end of the day. This permits responsible personnel to evaluate the system

7. *Open-Ended Design*

The resources management control program should have an open-ended design for ease of system expansion and modification. This will reduce the impact of changes to the control program on any existing subsystem or applications program.

### *System structure*

Figure 1 illustrates the hierarchial structure on three levels of multi-programming.

The first level utilizes the multi-programming capabilities of software supplied by the computer manufacturer. We call it the "host operating system." At this level the resources management subsystems will execute concurrently with other processors, such as COBOL, FORTRAN, or some other major application subsystems. The second level of multi-programming is accomplished by the regional supervisors (resources management subsystems) or the so-called "subtasking facility." This means that many independent programs (subtasks) can be run concurrently within a region or partition of the host operating system. The third level of multi-programming is achieved by means of reentrant module (or pure code). If a module will not self-modify any part of its code during processing, the module is capable of parallel processing of multiple numbers of transactions.

*System logic flow*

As an overview of the P. G. and E.'s corporate information system refer to Figure 2. You will notice that the manufacturer's operating system facility serves as the host to control the overall operation. On the left of the figure is the front end of the teleprocessing subsystem performing such functions as polling, addressing of the terminal network, message queuing and dispatching, terminal hardware error and security checking, etc. The next major part of the TP subsystem is the input message editing (TPCHUG) and output message editing package. This package makes all the message processing programs independent of terminal hardware. At the base of the figure, the file management subsystem[4] centralizes control of all I/O functions; it exchanges information from the information system data base and therefore, makes the data independent of applications programs (or message processing programs). Beneath the operating system we find the resources management subsystem functioning as the regional supervisor for all the subtasks within the on-line region and, therefore, we can support different application subsystems in the same region. We can also open another region for on-line message processing and share the file management, teleprocessing and resources management subsystems.

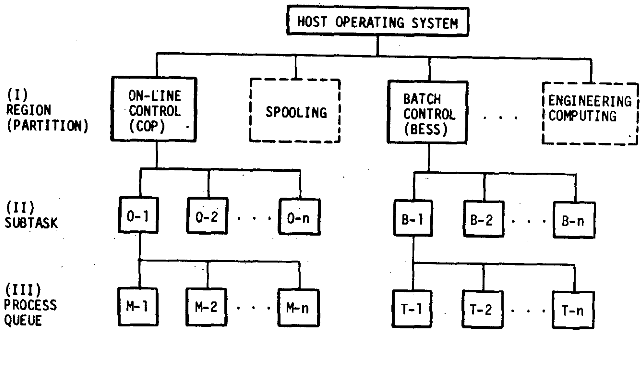


Figure 1—Hierarchical control on three levels of multiprogramming

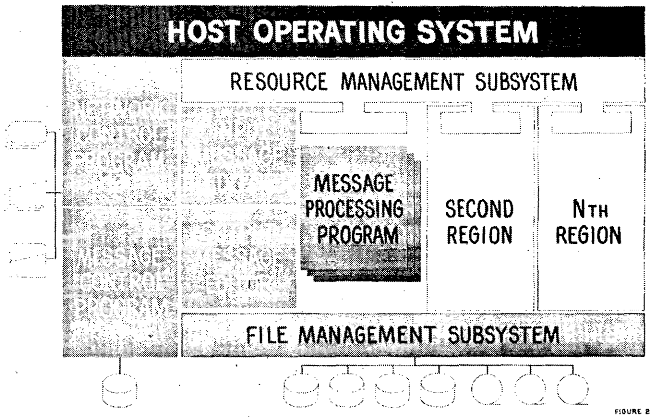

HOST OPERATING SYSTEM
RESOURCE MANAGEMENT SUBSYSTEM
MESSAGE CONTROL PROGRAM
MESSAGE PROCESSING PROGRAM
SECOND REGION
NTH REGION
FILE MANAGEMENT SUBSYSTEM
FIGURE 2

Figure 2—Host operating system

On the right side of the figure, we can open a number of background regions for computing or batch type jobs, or we can have batch executive subsystem (BESS) control the subtasking in the batch region. The rest of this section will briefly describe how a typical on-line region or a batch region works under the resources management subsystem.

## The on-line system

The Controller of On-Line Processing (COP) is designed to control the execution of the on-line portion of the system. Figure 3 graphically depicts how the computer's main memories are allocated during the on-line processing day.

- Initialization of the On-Line System

  At the start of each on-line day, two subsystems must be initialized. COP coordinates the actions of these two subsystems in preparation for the on-line execution of message processing programs.

  - At first, the teleprocessing subsystem must open the lines to the remote terminals and initialize the queues where the messages will await their turn to be transmitted and processed.
  - In the meantime, the initialization routines of COP are busy establishing the necessary control blocks and data pools (see Appendix I).
  - A program control block will be built for each message processing program. A master control block will be built for each transaction response pool (TRP). The number of TRPs to be used is specified by a parameter that is brought in with other start-up control data such as the number of event control blocks (Figure 4).
  - Subsequently, COP will cause the file management subsystem to build file/data control

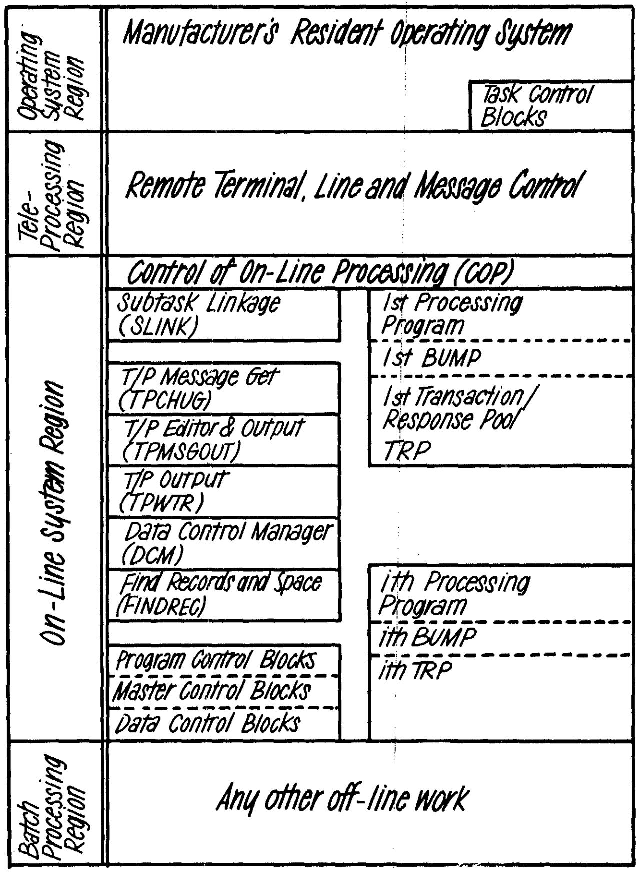


Figure 3—Main memory usage during on-line processing

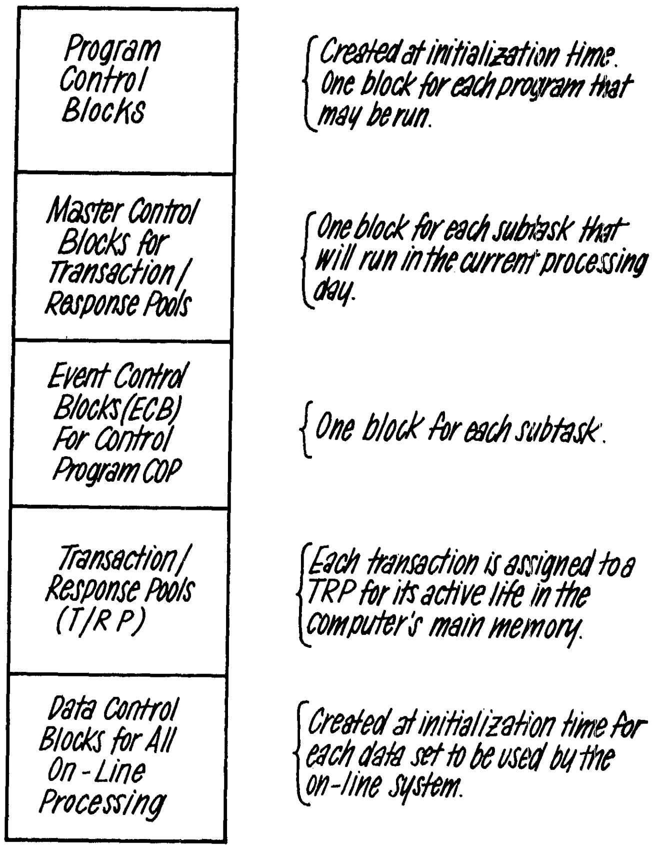

Program Control Blocks
Created at initialization time. One block for each program that may be run.
Master Control Blocks for Transaction/Response Pools
One block for each subtask that will run in the current processing day.
Event Control Blocks (ECB) For Control Program COP
One block for each subtask.
Transaction/Response Pools (T/R P)
Each transaction is assigned to a TRP for its active life in the computer's main memory.
Data Control Blocks for All On-Line Processing
Created at initialization time for each data set to be used by the on-line system.

Figure 4—Control blocks for on-line processing

blocks, record descriptors, and data buffer pools. When all the on-line files are opened and ready for processing, the file management subsystem will notify COP, and COP in turn will wait until the teleprocessing subsystem has completed its initialization processes.

- As soon as the teleprocessing routines have signaled their start of processing, COP repeats a sequence of actions to prepare message processing subtasks for multi-thread on-line operation. COP does this by establishing one subtask for each TRP and by signaling the operating system to start the teleprocessing get-a-message program (TPCHUG).
- Figure 5 shows a path from COP to TPCHUG, but this path is not taken until all subtasks have been established. For example, assume that five streams of messages will be processed concurrently. COP will signal the operating system five times that a subtask is to be established starting with TPCHUG, and before releasing control to any of these subtasks, COP will again signal the operating system that five more subtasks are to be established starting with Subtask **Link**age (SLINK). All ten of these subtasks are conditionally executable based on ten event control blocks which will be marked by COP. COP marks the five event control blocks for TPCHUG to show that the five associated TRPs are free, ready to accept input messages. SLINK's event control blocks are marked so that SLINK will wait until a message in a TRP is ready to be processed.
- COP will now wait for any one of five event control blocks to be marked by TPCHUG. It is while COP is waiting that Path 1 is taken and TPCHUG gets control for the first time; and the system is now active.

- Active On-Line System

TPCHUG makes use of the system support routine FINDREC to locate the place in the TRP

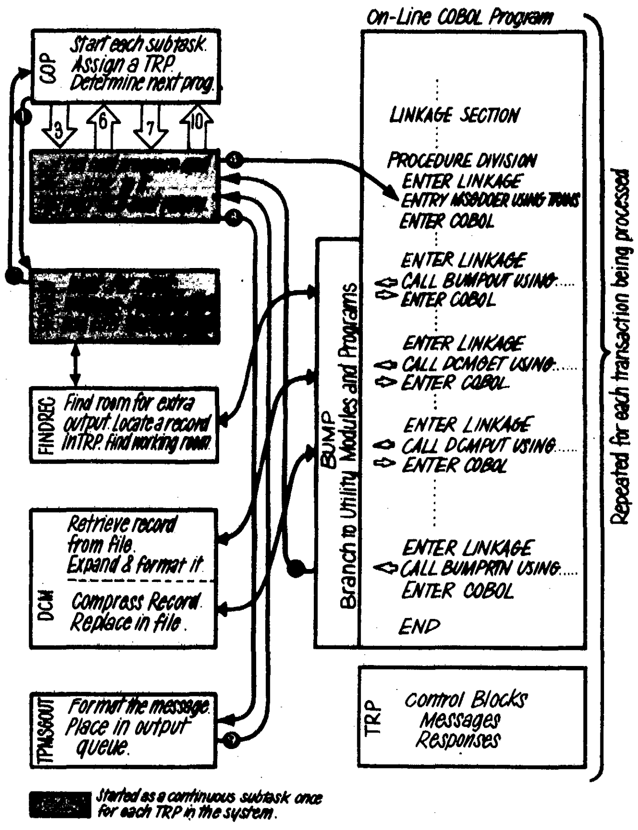


Figure 5—Flow of program control during on-line processing

where the incoming message and its internal format are to be placed. TPCHUG will do the following things:

- Get an incoming message from the waiting queue if one is available. If it is not, it will wait until there is one so that other subtasks may be executed.
- Place the message in the TRP.
- Edit and translate the message into the internal format.
- Mark COP's event control block for this TRP to show that a message is available for processing.
- Wait until TPCHUG's event control block is again posted by COP to show the TRP is again ready to accept a message.

When TPCHUG waits, Path 2 back to COP is effectively taken. COP will then check its event control blocks to determine which TRP requires service. The following actions are taken by COP:

- Via a program control list constructed by TPCHUG, COP will determine the next module to be applied to the current message in a TRP.
- Mark SLINK's event control block for this TRP to show that action is to be taken.
- Wait for the event control blocks to be marked by TPCHUG or SLINK.

Path 3 is now completed and SLINK will gain control when this subtask is made active. At this time the following functions will be performed:

- The required applications program will be loaded, if it is not already in core memory.
- Control will be given to the appropriate program so that it may execute.
- Upon return from the program, SLINK will mark COP's event control block to show that the program has completed its processing.
- Wait upon its event control block for this TRP.

Paths 4, 5, and 6 have been taken, and the same sort of thing occurs for Paths 7, 8, 9 and 10. The main difference is that when TPMSGOUT has finished putting the response(s) on the output waiting queue, COP's event control block is complete and the TRP is now free to be used again.

- Termination of the On-Line Day

Messages may be in waiting queues or in various stages of processing when termination of the on-line system occurs. COP must assure that the teleprocessing subsystem has received all incoming messages, the input waiting queues have all been emptied and all messages have completed processing before the file management routines close the files and COP releases the subtasks. While the teleprocessing programs are emptying the output waiting queues of messages and transmitting them, COP is editing the statistics which have been gathered that day and producing a report from them. After all processing has been completed, control is returned to the operating system.

## Batch system

- Initialization

Control information must be gathered and set up in main storage to effect the proper sequence of jobs to be run during a particular batch job stream. The aforementioned control information will contain such things as:

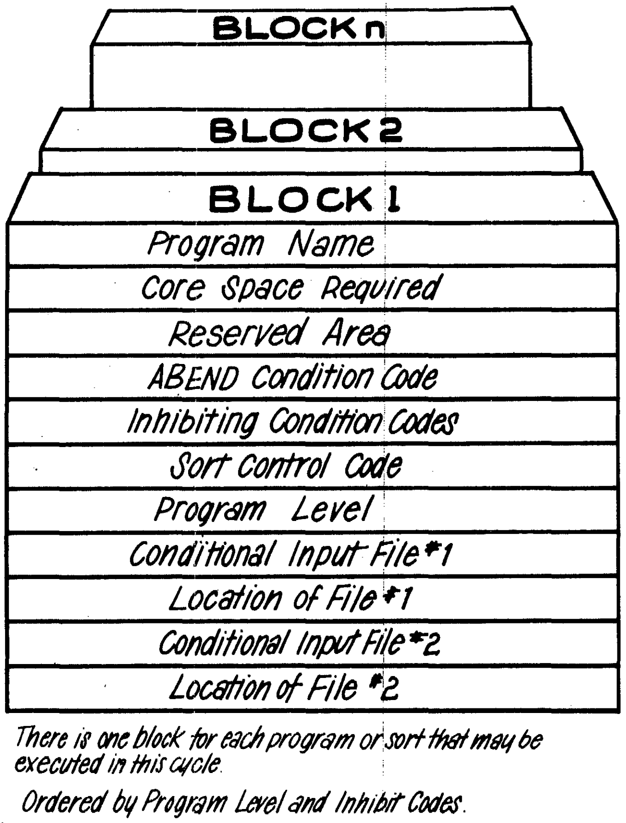
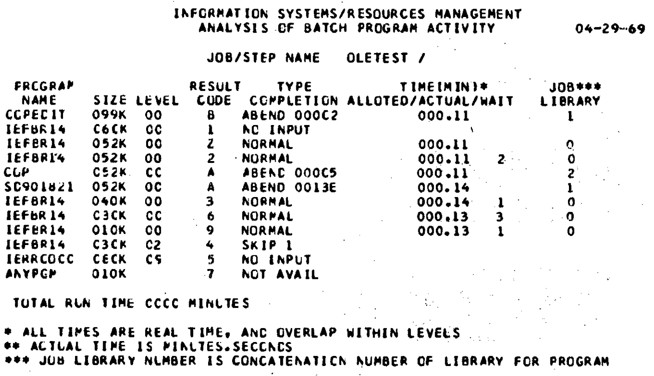

BLOCK n
BLOCK 2
BLOCK 1
Program Name
Core Space Required
Reserved Area
ABEND Condition Code
Inhibiting Condition Codes
Sort Control Code
Program Level
Conditional Input File #1
Location of File #1
Conditional Input File #2
Location of File #2
There is one block for each program or sort that may be executed in this cycle.
Ordered by Program Level and Inhibit Codes.

Figure 6—BESS program control blocks

- Program Name.
- Resultant condition code, if job fails to run.
- Condition code or codes resulting from other jobs which this job depends on in order to know whether or not to execute.
- Program level denoting whether or not this job can run concurrently with other jobs based on core usage, sequence of jobs to run and shared use of data files.
- Names and device locations of conditional files; i.e., files which might or might not be present during any one job within a job stream.

- Processing
During the so-called batch processing, depending on the information supplied for control during initialization time, jobs will be scheduled either alone or as subtasks, depending on core requirements, availability of data files, and shared access of data files.

- Termination
Upon completion of all the jobs which could be processed, the operator or other appropriate per-



```
                    INFORMATION SYSTEMS/RESOURCES MANAGEMENT
                      ANALYSIS OF BATCH PROGRAM ACTIVITY              04-29-69

                       JOB/STEP NAME   OLETEST /

 PRCGRAM                  RESULT    TYPE            TIME(MIN)*          JOB***
  NAME     SIZE LEVEL   CODE  COMPLETION ALLOTED/ACTUAL/WAIT    LIBRARY
CCPECIT    099K  00       8   ABEND 000C2          000.11               1
IEFBR14    C6CK  OC       1   NC INPUT
IEFBR14    052K  00       2   NORMAL               000.11               0
IEFBR14    052K  00       2   NORMAL               000.11    2          0
CUP        C52K  CC       A   ABENC 000C5          000.11               2
SC901821   052K  OC       A   ABEND 0013E          000.14               1
IEFBR14    040K  00       3   NORMAL               000.14    1          0
IEFBR14    C3CK  CC       6   NORMAL               000.13    3          0
IEFBR14    010K  00       9   NORMAL               000.13    1          0
IEFBR14    C3CK  C2       4   SKIP 1
IERRCOCC   CECK  C5       5   NO INPUT
ANYPGM     010K           7   NOT AVAIL

 TOTAL RUN TIME CCCC MINUTES

* ALL TIMES ARE REAL TIME, ANC OVERLAP WITHIN LEVELS
** ACTUAL TIME IS MINUTES.SECCNCS
*** JOB LIBRARY NUMBER IS CONCATENATICN NUMBER OF LIBRARY FOR PROGRAM
```

Figure 7—BESS status report

sonnel will be notified as to the status of all the jobs. Such information as:

- Jobs which were not run due to data not being available.
- Jobs which were not run to completion due to a program failure.
- Jobs which were run to completion.
- Other information which will help key personnel to effectively evaluate their choice of action.

*System facilities*

This section will describe some system support facilities implemented to enhance the reliability of the system.

### On-line system's facilities

- Program Test Mode of Operation

Any new or updated version of a message processing program which is to be added to the live system is first run in the test mode even though the program has gone through unit system and high level testing.

If it is an updated version of an operation program, it will have the last copy available as backup on the library. If it is to update files it will not be permitted to do so directly, but updated records will be saved, verified, and then added to the files in batch runs. Should the program fail during the test the old reliable copy will be brought in and normal processing will continue.

With new programs, no backup copy is available so if it fails, it is dropped from the system. A new program also affects the rest of the system in that while it is executing, any production programs that might fail will be refreshed and information processed by the new program will also be included in the output.

- Trouble Procedures

Trouble procedures and error defaults have been developed.

- Excess Elapsed Time

Should a program be in memory too long (elapsed time) to process a message, its priority will be raised so that it can get a greater share of the CPU's time.

- Excess Processing Time

Before control is turned over to a processing program, a timer is set by SLINK to the number of machine cycles to be allocated to this program for its processing. The number of machine cycles allocated is based upon operating history plus a safety factor. If a program should exceed the number of machine cycles allocated to it, an interrupt will occur and control will be returned to COP.

When a program is terminated due to excessive operating time or has abnormally ended, a snapshot of the TRP and all associated control blocks is taken. An error message is returned to the terminal from where the transaction was entered and the control block for the program is marked to show that an error has occurred. If the program is used by more than one type of transaction, or is critical to system operation, a fresh copy will be brought into memory. If the program is not normally resident, a fresh copy will be brought in before it is used again.

If a program that exceeds the processing time is running in the test mode, it will not be used again, but a backup version will be used instead. If two successive errors occur, the program will be marked unavailable and the type of transaction that precipitated the error will be rejected until manual action is taken to reinstate them.

- Accumulation of Statistics

During the operation of the on-line system, COP is gathering statistics about the various programs. These statistics are recorded in the appropriate program control block and a report produced at the end of the day (Figure 8). COP is also recording each TRP so that they may be analyzed after the day's run.

## Batch Executive Sub-System (BESS)

- Improved Throughput

Some operational batch jobs that ran under 360-OS-MVT (Multi-programming with Variable number of Tasks) were compared to multi-tasking under BESS and MVT. They showed a 60 percent savings of time under BESS.

This can be attributed to:

- Subtasking

By running two or more of the jobs concurrently as subtasks to BESS within an MVT region, it then makes it possible to take the greatest advantage of the computer's resources.

- Reduction of Job Control Information

All the jobs are run as subtasks to BESS and not as individual jobs. This eliminates the need for a great deal of the job control language cards for each program. The control information required by BESS is minimal and takes less time and space to read, interpret, and use (Figure 6). Only one control record per program is required.

- Sharing of Record Accesses

Each time a record is read from a direct access or sequential storage device, it can be shared by two or more programs that are run as concurrent subtasks to BESS. There are restrictions such as only one program may update the file and all must look at the records in the same order or fashion. This savings of record access times is appreciable for large files in a corporate data bank.

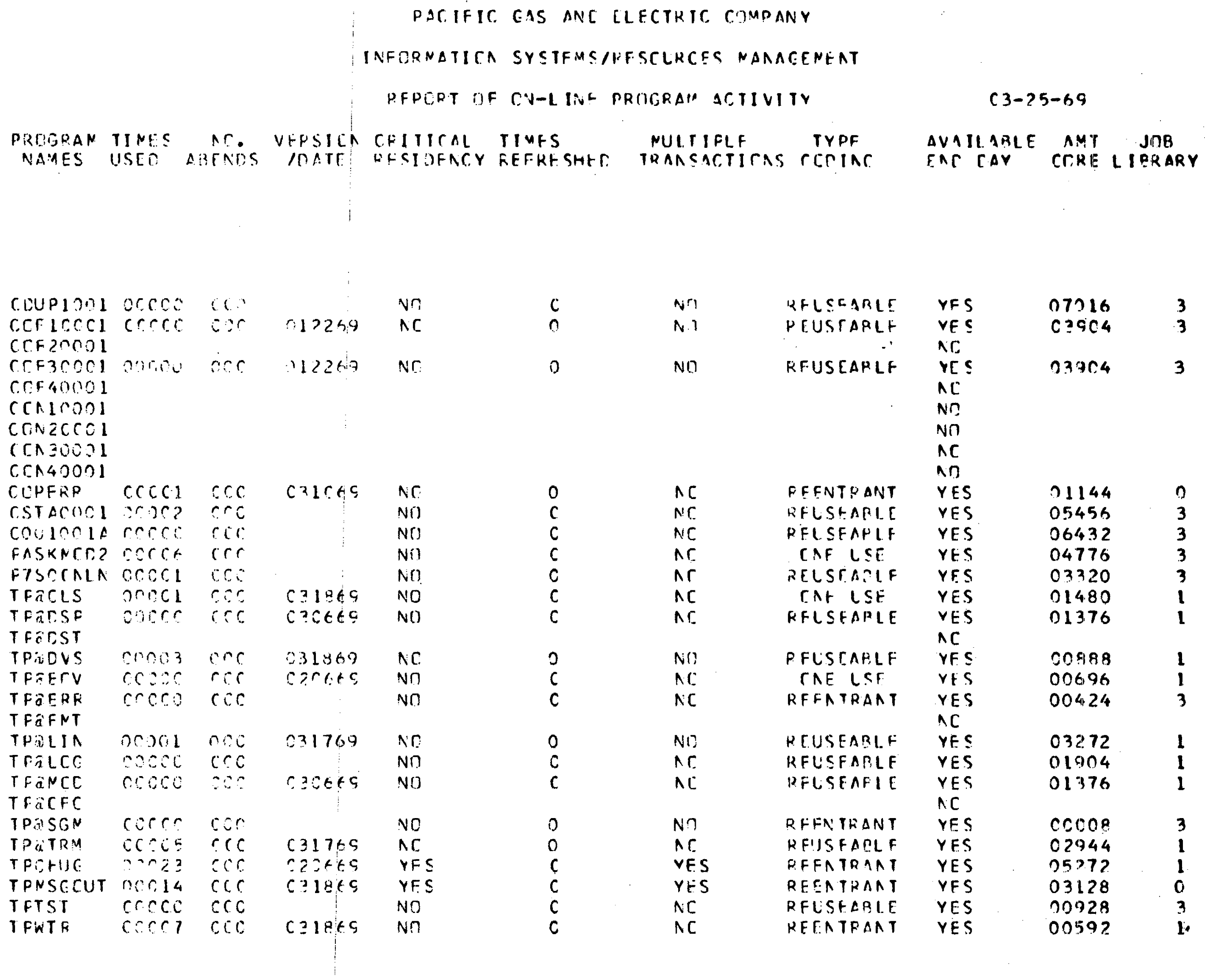
- Job Scheduling

The programs to be run are dependent upon the data that was collected during the on-line day. BESS will determine which programs are to be run and in what order.

- Conditional Runs

Part of the control information supplied BESS is the input file (or files) which must contain data if the program is to be executed. If data is present, BESS will schedule the program for execution. If the data is not present, BESS will not schedule the program.

Another part of the control information is a



```
                                   PACIFIC GAS AND ELECTRIC COMPANY

                                INFORMATION SYSTEMS/RESOURCES MANAGEMENT

                                   REPORT OF ON-LINE PROGRAM ACTIVITY                    03-25-69

PROGRAM  TIMES   NO.    VERSION CRITICAL  TIMES        MULTIPLE       TYPE         AVAILABLE  AMT    JOB
 NAMES   USED  ABENDS   /DATE  RESIDENCY REFRESHED  TRANSACTIONS   CODING        END DAY   CORE LIBRARY




CDUP1001 00000  000               NO         0           NO        REUSEABLE      YES     07016      3
CCF10001 00000  000    012269     NO         0           NO        REUSEABLE      YES     03904      3
CCF20001                                                                          NO
CCF30001 00000  000    012269     NO         0           NO        REUSEABLE      YES     03904      3
CCF40001                                                                          NO
CCN10001                                                                          NO
CCN20001                                                                          NO
CCN30001                                                                          NO
CCN40001                                                                          NO
CDPERP   00001  000    031069     NO         0           NO        REENTRANT      YES     01144      0
CSTA0001 00002  000               NO         0           NO        REUSEABLE      YES     05456      3
COU1001A 00000  000               NO         0           NO        REUSEABLE      YES     06432      3
FASKMCD2 00006  000               NO         0           NO         ONE USE       YES     04776      3
F7SOCNLN 00001  000               NO         0           NO        REUSEABLE      YES     03320      3
TP@CLS   00001  000    031869     NO         0           NO         ONE USE       YES     01480      1
TP@DSP   00000  000    030669     NO         0           NO        REUSEABLE      YES     01376      1
TP@DST                                                                            NO
TP@DVS   00003  000    031869     NO         0           NO        REUSEABLE      YES     00888      1
TP@ECV   00000  000    020669     NO         0           NO         ONE USE       YES     00696      1
TP@ERR   00000  000               NO         0           NO        REENTRANT      YES     00424      3
TP@FMT                                                                            NO
TP@LIN   00001  000    031769     NO         0           NO        REUSEABLE      YES     03272      1
TP@LOG   00000  000               NO         0           NO        REUSEABLE      YES     01904      1
TP@MCD   00000  000    030669     NO         0           NO        REUSEABLE      YES     01376      1
TP@OFC                                                                            NO
TP@SGM   00000  000               NO         0           NO        REENTRANT      YES     00008      3
TP@TRM   00005  000    031769     NO         0           NO        REUSEABLE      YES     02944      1
TPCHUG   00023  000    020669     YES        0           YES       REENTRANT      YES     05272      1
TPMSGOUT 00014  000    031869     YES        0           YES       REENTRANT      YES     03128      0
TPTST    00000  000               NO         0           NO        REUSEABLE      YES     00928      3
TPWTR    00007  000    031869     NO         0           NO        REENTRANT      YES     00592      1
```

Figure 8—COP status report

set of inhibit codes for the program. Each program that does not run correctly to completion will cause a condition code to be set and if the particular combinations of inhibit codes for a program are satisfied, it will not be run. This saves the time of loading and attempting to execute a program that should not be executed.

- Maximun Use of Resources

Each program's control information contains a level code which shows two things. The first is the order of execution. A program with a level code of three cannot be run until all programs with level codes less than three have been run or rescheduled. The second function of the level code is to designate groups of programs which may be run concurrently without conflict.

With this information, BESS can execute as many subtasks as the computer's resources permit.

- Other Features

This approach to controlling batch runs has two other major assets.

- Better Error Control

If a program abnormally terminates, BESS will regain control and can invoke error procedures to salvage as much as possible from the run. By doing so, all other parallel jobs can continue running without interruption and most of the work can still be done while a fix is being implemented to correct the error.

- Piggy Back Programs

  There are a variety of programs that are run only once or very rarely. These programs do not update the files but only look at the records to perform some analysis. These programs can ride piggy back on some other run in the system under BESS. The added time is negligible compared to the cost of passing the voluminous files used in a large corporate information system.

- Reduction of Testing Time for Application Subsystem

  An application subsystem within a corporate information system often contains a goodly number of interrelated program modules.

  During the system integration testing, if only one of the modules fails, it will cause the whole job to fail. Therefore, to test all the modules in a combined environment can be very time-consuming.

  However, under BESS each module can be classified as a subtask. If a subtask fails, the entire job will not terminate and BESS can proceed to test the remaining modules. Figure 7 shows a report by BESS to facilitate the analysis of the testing.

*System Implementation*

The Resources Management Programs have been written in IBM-360 Operating System Assembly Language (ALC).

1. It is fully interfaced with the IBM-360 operating system MVT.
   - The applications programs and the systems programs operate as independent subtasks of the regional resource manager; abnormal termination of a subtask will not stop the remaining subtasks in the region.
   - The package is not tied to any particular release of O/S; hence, if a new version is released, there should be little effect on this package.
2. The Resources Management packages take full advantage of existing operating system facilities and make extensive use of the subtasking and master scheduler facilities.
3. It is intended to interface with all the operating system supported languages (COBOL and ALC interface have been implemented).
4. The entire package has been designed to be dynamic in nature; that is, all programs are load modules. They are not linkage edited into the applications program; thus, the package may be redesigned and improved without any appreciable effect on the applications programs.
5. The entire package has been programmed in re-entrant code.
6. The hardware anticipated over the next several years includes two large central processors with a million bytes of main memory, supported by smaller satellite computers and a score of multi-drive disk storage units. The system is being designed to support several hundred terminals, most of which are expected to be high speed CRT display units.

APPENDIX I

- *On-Line System Control Blocks*

The on-line system makes use of the control blocks established during initialization time. These blocks are:

- Program Control Block (Figure 9)

  This control block contains the program identification, the storage address within the library, operating system controls, on-line control data, and counters to record the number of times the program was used. It permits the on-line system to bring a needed program in from the library in an efficient manner. This block is also used by COP to collect statistics about the program and determine its current status.

- The Master Control Block for the TRP (Figure 10)

  This block contains pointers to various other control blocks as well as containing the two event control blocks which apply to the subtasks associated with the TRP. One pointer shows the location of the TRP, one points to the task control block located in the operating system's region, and another to the event control block that will be marked to tell COP when a program has completed processing in this TRP .The next one can either be a pointer to the processing program or its identifier depending upon conditions at the time. Two event control blocks are next

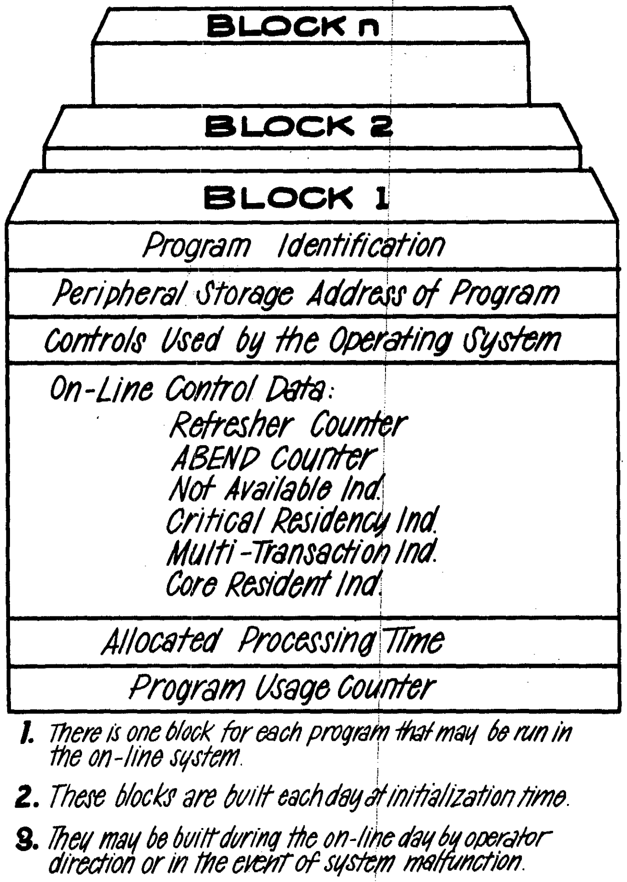

BLOCK n
BLOCK 2
BLOCK 1
Program Identification
Peripheral Storage Address of Program
Controls Used by the Operating System
On-Line Control Data:
Refresher Counter
ABEND Counter
Not Available Ind.
Critical Residency Ind.
Multi-Transaction Ind.
Core Resident Ind.
Allocated Processing Time
Program Usage Counter
1. There is one block for each program that may be run in the on-line system.
2. These blocks are built each day at initialization time.
3. They may be built during the on-line day by operator direction or in the event of system malfunction.

Figure 9—Program control blocks used by the controller of on-line processing (COP)

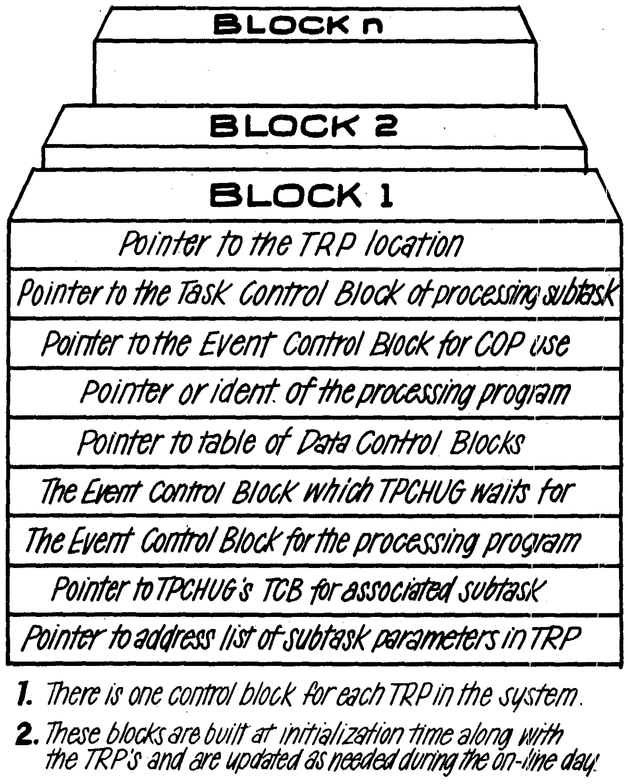

BLOCK n
BLOCK 2
BLOCK 1
Pointer to the TRP location
Pointer to the Task Control Block of processing subtask
Pointer to the Event Control Block for COP use
Pointer or ident. of the processing program
Pointer to table of Data Control Blocks
The Event Control Block which TPCHUG waits for
The Event Control Block for the processing program
Pointer to TPCHUG's TCB for associated subtask
Pointer to address list of subtask parameters in TRP
1. There is one control block for each TRP in the system.
2. These blocks are built at initialization time along with the TRP's and are updated as needed during the on-line day.

Figure 10—Master control blocks for transaction/response pools (TRP's)

and they are the ones used by TPCHUG and SLINK, respectively. The next pointer is to TPCHUG's task control block and the last to a list of addresses which in turn point to the parameters and records to be used or updated by the message processing program.

The Transaction/Response Pools (Figure 11)

These pools of data and controls are the heart of the on-line system. Their number determines how many messages can be processed concurrently. The first part of the TRP contains control information to help FINDREC in locating the various records and spaces within the pool. These controls also help COP to locate programs and other control blocks that are required. Because COP is re-entrant, it cannot store information in an area reserved for itself, so all controls for a given message must be maintained in the TRP apart from all others; therefore, no "cross talk" will occur between concurrently processing subtasks.

The latter part of the TRP is used for the message, its response(s) and overflow control if more room should be needed. Currently, 4096 bytes of memory for a TRP have been adequate to handle 98 percent of the messages processed. Expansion is provided by going to an overflow area if more room is needed. Figure 12 shows the relationship between the control blocks and TRPs.

## GLOSSARY

*BESS*

**B**atch **E**xecutive **S**ub-**S**ystem

*BUMP*

**B**ranch to **U**tility **M**odules and **P**rograms
This is a small module appended to each high-level language program to:

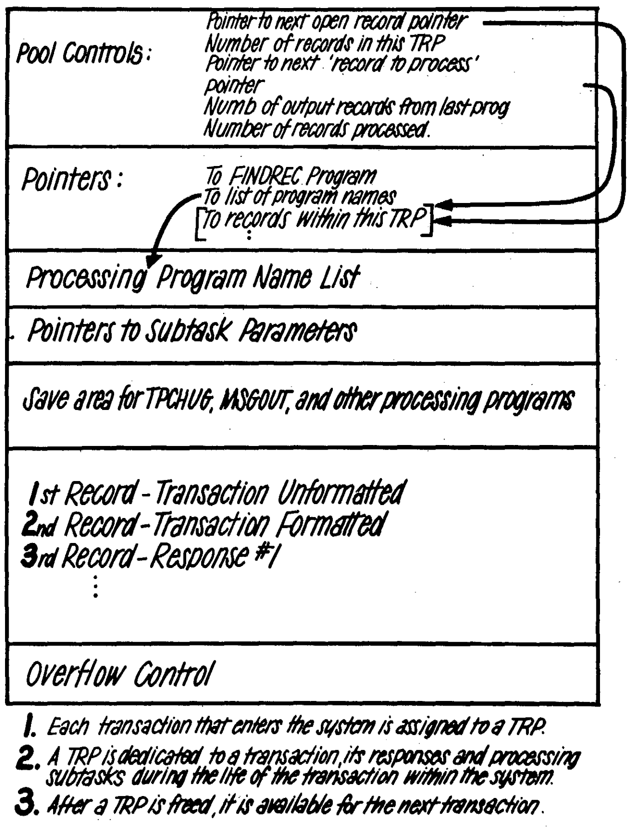

Pool Controls:
Pointer to next open record pointer
Number of records in this TRP
Pointer to next 'record to process' pointer
Numb of output records from last prog
Number of records processed.
Pointers:
To FINDREC Program
To list of program names
To records within this TRP
Processing Program Name List
Pointers to Subtask Parameters
Save area for TPCHUG, MSGOUT, and other processing programs
1st Record - Transaction Unformatted
2nd Record - Transaction Formatted
3rd Record - Response #1
Overflow Control
1. Each transaction that enters the system is assigned to a TRP.
2. A TRP is dedicated to a transaction, its responses and processing subtasks during the life of the transaction within the system.
3. After a TRP is freed, it is available for the next transaction.

Figure 11—Transaction/response pool (TRP)

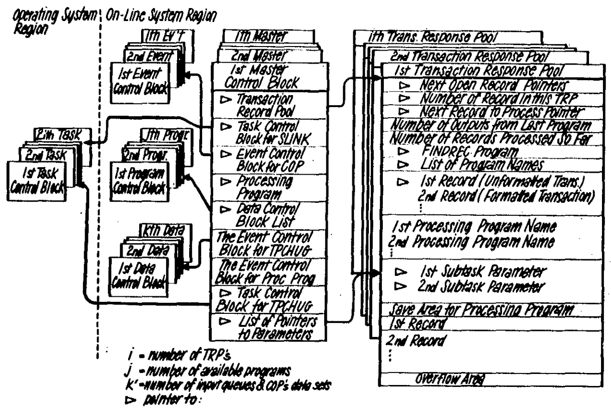

Operating System Region
On-Line System Region
ith Ev't
2nd Event
1st Event Control Block
ith Master
2nd Master
1st Master Control Block
▷ Transaction Record Pool
▷ Task Control Block for SLINK
▷ Event Control Block for COP
▷ Processing Program
▷ Data Control Block List
The Event Control Block for TPCHUG
The Event Control Block for Proc Prog
▷ Task Control Block for TPCHUG
▷ List of Pointers to Parameters
ith Task
2nd Task
1st Task Control Block
jth Progr.
2nd Progr.
1st Program Control Block
kth Data
2nd Data
1st Data Control Block
ith Trans. Response Pool
2nd Transaction Response Pool
1st Transaction Response Pool
▷ Next Open Record Pointers
▷ Number of Record in this TRP
▷ Next Record to Process Pointer
Number of Outputs from Last Program
Number of Records Processed So Far
▷ FINDREC Program
▷ List of Program Names
▷ 1st Record (Unformatted Trans.)
2nd Record (Formatted Transaction)
1st Processing Program Name
2nd Processing Program Name
▷ 1st Subtask Parameter
▷ 2nd Subtask Parameter
Save Area for Processing Program
1st Record
2nd Record
Overflow Area
i = number of TRP's
j = number of available programs
k' = number of input queues & COP's data sets
▷ pointer to:

Figure 12—Control block relationship in the on-line system

- find the various system support modules within a dynamic environment.
- establish the linkage between the program and the system support module.

*COP*

Controller of On-Line Processing

*DCM*

Data Control Manager
A subsystem which controls the data base for the Management Information System.

*DCMGET*

An on-line DCM subroutine which retrieves selected information from the data base for applications programs. It makes the applications programs independent of the data base structure.

*DCMPUT*

An on-line DCM subroutine which selectively updates the data base for applications programs.

*ECB*

Event Control Block
A small block of memory that contains indicators to show if a program is waiting for an event to occur, if the event has occurred and the completion code associated with the event, when posted.

*FINDREC*

A resources management on-line subroutine that does dynamic space allocation and locates data within the transaction/response pools.

*Multi-tasking*

The interleaved or time-shared execution of two or more program tasks within a single CPU; multiprogramming.

*On-Line*

Pertaining to the responsiveness of a computer system. To respond in a timely fashion to user's needs who have direct access to the computer via data entry devices, terminals and displays; real-time.

*SLINK*

Subtask Linkage
An on-line resources management program that is a constant subtask to COP and links to each processing program.

*Subsystem*

A system of interrelated programs that is subordinate in control and execution to another system.

*Subtask*

An executable program that has all the attributes of a task but is subordinate to and under the control of another task.

*Task*

One of two or more programs, or series of programs which execute concurrently in a single CPU.

*TPCHUG*

A teleprocessing program that is a constant subtask to COP. It reads transactions from the input waiting queue, edits them, and translates them into their processing format.

*TPMSGOUT*

A teleprocessing subroutine that converts a response to a terminal's format and places it into the output waiting queue.

*TRP*

**T**ransaction **R**esponse **P**ool
A block of memory which contains a single raw transaction (message), some of its control information, its intermediate forms and its response(s). A TRP is assigned to one transaction at a time for its active life within the CPU. It contains all data associated with the transaction in chronological sequence so it is useful for debugging.

## ACKNOWLEDGMENT

The authors wish to thank Mr. J. R. Kleespies for his encouragement and support; Mr. W. D. Ayers and Mr. R. T. St. Germain for their dedicated efforts in design and programming; Miss Agnes Wolf and Mrs. L. J. Fiore for their meticulous typing; Mr. Leo Karl of IBM for initial analysis and Messrs. F. J. Thomason and J. W. Nixon of Haskins & Sells for their invaluable advice.

## REFERENCES

1 S ROSEN
*Electronic computers: A historical survey*
ACM Computing Surveys Vol 1 No 1 March 1969

2 R F ROSIN
*Supervisory and monitor systems*
ACM Computing Surveys Vol 1 No 1 March 1969

3 J DIEBOLD
*Thinking ahead: Bad decision on computer use*
Harvard Business Review Jan-Feb 1969

4 H LIU
*A file management system for a large corporate information system data bank*
Proc FJCC Vol 33 1968

The following references are selected by the authors for general background information, but are not mentioned in the text.

5 J MARTIN
*Programming real-time computer systems*
Prentice-Hall 1965

6 J MARTIN
*Design of real-time computer systems*
Prentice-Hall 1967

7 M G JINGBERG
*Notes on testing real-time systems programs*
IBM Systems Journal Vol 4 No 1 1965

8 J D ARON
*Real-time systems in perspective*
IBM Systems Journal Vol 6 No 1 1967

9 J W HAVENDER
*Avoiding deadlock in multitasking systems*
IBM Systems Journal Vol 7 No 2 1968

10 B I WITT
*Job and task management*
IBM Systems Journal Vol 5 No 1 1966

11 D D KEEFE
*Hierarchical control programs for systems evaluation*
IBM Systems Journal Vol 7 No 2 1968

12 W C McGEE
*On dynamic program relocation*
IBM Systems Journal Vol 4 1965

13 *IBM system/360 operating system*
MVT Control Program Logic Summary Form Y28-6658

14 *IBM system/360 operating system*
MVT Supervisor Form Y28-6659

15 *IBM system/360 operating system*
MVT Job Management Form Y28-6660

16 *IBM system/360 operating system*
Supervisor and Data Management Services Form C28-6646

# Incorporating complex data structures into a language for social science research

*by* STEPHEN W. KIDD

*The Brookings Institution*
Washington, D. C.

## INTRODUCTION

This paper presents a set of augmentations to the language BEAST* (Brookings Economics and Statistical Translator) as part of a continuing effort to define a language for a particular group of computer users, social scientists. In this nebulous group we include professional economists, political scientists, psychologists, sociologists, and a large number of university students in those disciplines. An important assumption underlying our work has been that the cost of not having substantially better software than presently exists is very large and should be measured in terms of researchers' time. The true cost of inappropriate methods of computer utilization should not be measured by staff and computer costs, but by the social cost of the output foregone. When answers to questions of importance for national public policy formation require weeks, months, or even years to obtain, the cost becomes a social cost that we all eventually bear.

BEAST is a computer language designed to embody many of the concepts of the more quantifiable social sciences. The present version of the BEAST operates primarily upon "rectangular" data files, that is, files having observations on attributes of enumeration units. In other words, acceptable files consist of fixed length logical records, one record for each enumeration unit. Many social science data files either have this structure or can be cast in this structure with little effort, and the majority of "general purpose programs," written for social scientists also assume this data structure. However, many social science data files have a more complex structure and cannot be processed either by the present version of the BEAST language or by most existing computer programs.

This paper describes possible extensions to the BEAST language to make it useful for processing data with a more complicated structure. Though the data structures and language constructs described here could be applied to extensions of other languages, we feel that they have particular utility when combined with features already available in the BEAST. The intent of the proposed extensions is not to introduce a general list processing capability into the language as has been done with some other languages,[6,9,16,18] but to accommodate a particular class of files characterized by hierarachical record structures. We have deliberately decided in favor of a limited structure that permits the ease of reference that is essential for the users we envision for the language. The generality of those complex structures which have been disallowed in the current proposal is a luxury which can only be bought for a significant price—the increased specificity required in a language to reference such structures. The user who wants such generality pays the price in other languages in the increased tedium of writing his program.

* Jeffrey W. Bean, Stephen W. Kidd, George Sadowsky, Beverly D. Sharp, *THE BEAST: A User-Oriented Procedural Language for Social Science Research.* (The Brookings Institution, June 13, 1968). Reference to "the current BEAST" should be understood to refer to that paper.

Consider a slight variant of the 1966 Survey of Economic Opportunity (SEO) File constructed by the U. S. Office of Economic-Opportunity. The organization of data within each enumeration unit is tree-structured, that is, each level or segment of data may be followed by a variable number of segments of data at the next lower level. Figure 1 illustrates the structure of this file.

Disaggregation by respondent characteristics, time period, income group, geographic area or other conditions is often very fruitful for social science research. For example, using this file it should be possible to define a subset of households or families based upon person characteristics, or the reverse. Such groups might be (1) the set of all families such that no persons are 65 or more years old, (2) the set of all households such that at least two persons earn $5,000 or more per year in wages and salaries, (3) the set of all families such that exactly two persons are less than 21 years old, (4) the set of all persons whose households are headed by a woman, (5) the set of all families that live in the northeast, and (6) the set of all persons whose families are at least five persons in size and which live in the southwest.

The current BEAST language provides the DEFINE SAMPLE statement for defining a subset of the user's original population and the ON SAMPLE suffix for restricting computations to observations within that subset. The format of the DEFINE SAMPLE statement is:

DEFINE SAMPLE samplename AS logical
    expression

An example of a DEFINE SAMPLE statement would be

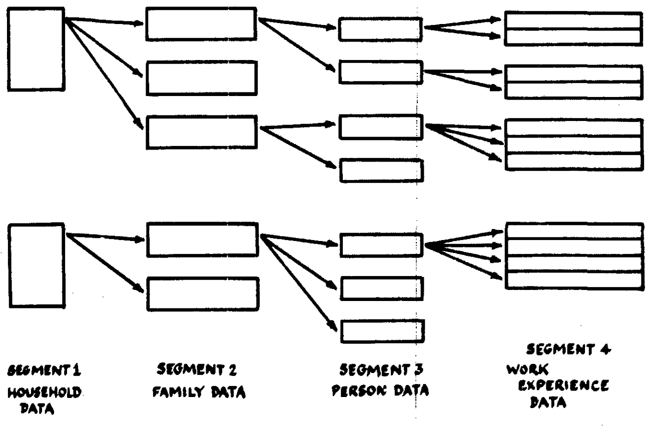


Figure 1—A logical structure for the survey of economic opportunity file

DEFINE SAMPLE OLDMEN AS AGE > 65
    AND SEX EQ 'M'

That sample definition could be invoked using the ON SAMPLE suffix to compute the average income of the old men in a set of data:

LET AVINC = MEAN (INCOME) ON
    SAMPLE OLDMEN

The ON SAMPLE suffix can be used in a similar way to define a restricted domain for calculation of derived variables, statistical procedures, and input and output.

While the current definition of the language is sufficient to express extremely general conditions on rectangular data files, the syntax for logical expressions is insufficient for defining samples of the type mentioned above for the SEO file. The next two sections describe an augmented I/O facility and an expanded conditional expression syntax designed to evaluate logical functions on data structures of the type indicated.

Before proceeding further, it is useful to formalize somewhat the data structure indicated in Figure 1. Data related to a single entity like a person, a family a state, or a company we shall call a *segment*.* An occurrence of a segment resembles one row of a rectangular data matrix: it is one set of values for a list of attributes, and it is defined by the list of attributes included in one occurrence of the segment. For example, a segment describing a person (a PERSON segment) might be defined by the list of attributes AGE, SEX, INCOME, and RENT. We denote that a PERSON segment is composed of values for those four attributes by writing

PERSON [AGE, SEX,INCOME, RENT]

or in general with the notation

segmentname [attributelist]

* The concept of a segment as described here should not be confused with its usage in discussions of virtual memories and address spaces. Our usage is close to what R. M. Balzer has called a "collection" in "Dataless Programming", (Rand Corporation July, 1967) Memorandum RM-5290-ARPA. It also resembles the usage in COLINGO of "group": *COLINGO C-10 User's Manual,* (Mitre Corporation, May 1968) Document ESD-TR-66-653; and the POP-2 usage of the term "record." R. M. Burns, J. S. Collins, "An Introduction to the POP.2 Programming Language," (University of Edinburgh, October, 1967,) Min-MAC Reports, No. 4. The term segment has been adopted for IBM's GIS file management system.

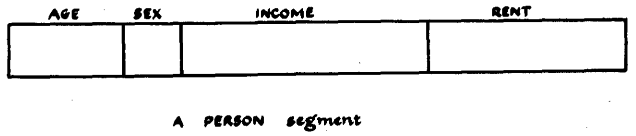


Figure 2—A PERSON segment

Figure 2 shown an example of one such PERSON segment. A rectangular matrix would be composed of a set of such "segments" conceptually placed one below the other.

As an extension of that structure, segments can be combined by linking them to construct a "tree". The tree has as its "root" a single segment, and has as its "branches" one or more different segments. Figure 3 shows one such tree structure representing one FAMILY and three PERSON's.* **

A tree such as in Figure 3 is the basic unit in our augmented data structure.

We call shall each successive tier of the data hierarchy a *level*. Levels are numbered and begin at one, the level for those segment types not contained in any other segment. Level one is the *highest* segment level possible. Every segment type has a unique level associated with it, though more than one segment type may occur at any level. When a segment S is connected to segment T by a single path through one or more segments, we shall say that S *contains* T (conversely, T *is contained* in S). All segments contained in segment S are called *subsegments* of S. Segments are contained in a unique segment of the next higher level. This restriction on the data structure permits simplification of the language we use to reference the structures. In particular, it permits attributes of segments at one level to be "imputed" to segments at a lower level, and it obviates explicit upward-references when referring to low level segments.

We shall call information about containment (which segments contain or are contained in which other segments) *structural information* about the data, as distinct from the data itself. The structural information of a file is often contained only implicitly in the physical arrangement of the data in the file. When data are read into memory, the structural information

* These figures give no indication of the physical structure of the data. There are several reasonable ways in which such data could be arranged, but the language used to talk about such data should be independent of the physical arrangement of the data.

** For convenience we will call "an occurrence of the structure defined by "X" simply "an X".

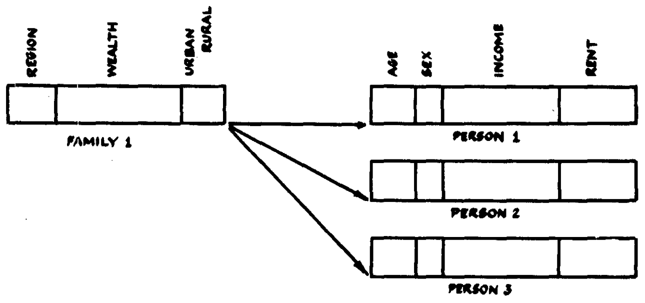


Figure 3—A simple enumeration unit

should become explicitly represented as a list structure for efficiency in processing.

Trees of the forms described above can often represent naturally the structure of the enumeration units, (EU's) encountered in social science research. For the purposes of this paper the tree that represents an enumeration unit consists of a unique segment type at level one called the *root segment* together with all its subsegments. A *file* is an ordered set of such enumeration units. To denote that an enumeration unit has a structure we shall give the entire aggregate a name and define its constituents according to their relations. The simple tree structure in Figure 3 would be defined in BEAST by writing

```
DEFINE EU FAMSTRUCT AS
  1 FAMILY [REGION, WEALTH,
    URBANRURAL]

  2 PERSON [AGE, SEX, INCOME,
    RENT]
```

The purpose of such a definition is to describe the set of possible occurrences of the enumeration unit, since an EU definition says nothing about whether a *particular occurrence* of the structure will actually have any subsegments, the number of subsegments, or the physical order or type of the attributes in the segments.

Another example of an EU definition is:

```
DEFINE EU CONGCOMMITTEE AS
    1 COMMITTEE [NAME, BUDGET]

    2 MEMBER [LAST, FIRST, STATE,
       PARTY]
```

This definition specifies a tree structure with two levels that represents a Congressional committee. The root segment is a COMMITTEE segment, and for the purposes of CONGCOMMITTEE it has only two attributes, NAME and BUDGET. Segments of type COMMITTEE are assumed to contain only segments of type MEMBER. On input only structural information relating COMMITTEE and MEMBER segments will be retrieved from the file though the file may contain other segment types and attributes. On output only the structural information indicated will be displayed. As a third example consider the structure defined by the statement

```
DEFINE EU DWELLING AS
        1 HOUSEHOLD [AGEOFHEAD,
                SEXOFHEAD]

        2 FAMILY

        3 PERSON [AGE, INCOME]
```

When a segment name (FAMILY) is included in an EU definition with no attributes listed, then only the structural information at that level is extracted from the file. In this example, DWELLING would have the form indicated in Figure 4. With such an EU structure one could evaluate logical expressions that required structural information, but no attributes, at the family level. We might, for example, reference

```
PERSON'S IN FAMILY'S WITH AT LEAST 4
        PERSON'S
```

No FAMILY attributes are needed because only

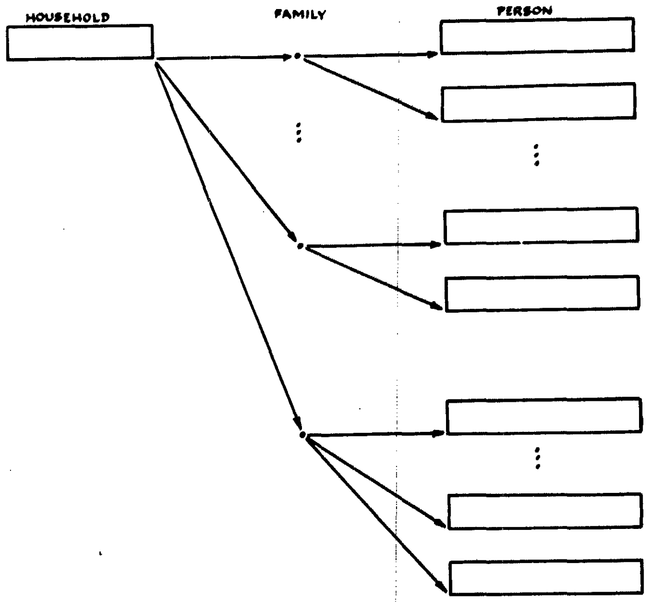


Figure 4—Example 3

structural information is required to evaluate this expression.

New attributes for FAMILY segments could also be generated from such a structure that begins with a null attribute list. We could compute the total income in each FAMILY segment (the sum of all PERSON'S income contained in that segment) using the TOTAL function:

```
LET FAMINC = TOTAL INCOME WITHIN
        FAMILY
```

The function TOTAL has the general form

$$\text{TOTAL}\left\{\begin{matrix}\text{attribute}\\ \\ \text{segmntid}\end{matrix}\right\}\left[\begin{matrix}\text{WITHIN segmntid}\\ \text{[subscript] [boolprim]}\end{matrix}\right]^{*}$$

In the example above we have taken the total of an attribute (INCOME), where the summation is taken over all values for INCOME contained within the specified segment, FAMILY. BEAST assumed that iteration is intended over all FAMILY segments since no subscript or modifier is put on the segment identifiers.

To explicitly assign a new attribute to a segment we will use the notation

```
LET segmntid: attributename = expression
```

Using this notation and the TOTAL function to count PERSON subsegments, we can create an attribute in each FAMILY segmente qual to theav erage income of all persons in the FAMILY:

```
LET FAMILY: AVINC = (TOTAL INCOME
                    WITHIN FAMILY)/
                    (TOTAL PERSON'S
                    WITHIN FAMILY)
```

Again iteration over FAMILY segments is implied because the segment identifier is unqualified. The value of the function would become a scalar if the second segment identifier were qualified with either a simple logical condition or a BEAST subscript.

For example, the statement

```
LET X = TOTAL INCOME WITHIN
        FAMILY'S (1...100)
```

* The syntactic type *boolprim* represents a single logical term.

would compute the sum of the income of all persons contained in the first 100 families and assign the value to the scalar variable X.

*Conditions on structures*

Logical expressions to deal with tree structures of the type described in the above section must be capable of expressing both intra-segment relations (analogous to present BEAST logical expressions) and interlevel relations among segments contained in or containing the reference segment of an expression. The reference segment of an expression is that segment with which the value of the expression is associated, distinguishing it from the other segments upon which the value of the expression may also depend. For example, the reference segment of the logical expression

FAMILY'S IN HOUSEHOLD'S WITH LESS THAN 10 PERSON'S

is the FAMILY segment because the expression defines a condition on FAMILY segments. Three segment types appear in the expression—FAMILY, HOUSEHOLD, and PERSON—but the value of the entire expression is clearly a condition on each FAMILY. Had the expression been simply

HOUSEHOLD'S WITH LESS THAN 10 PERSON'S

then the reference segment would have been HOUSEHOLD.

Table I gives a formal syntax for sample definitions using the proposed extensions to logical expressions. The set of words WITH, IN WHICH, etc., are used as "noise" words and are not significant for the interpretation of an expression. The construct *'S* is used optionally to imply a plural and not a possessive. Note that, for example, the plural of FAMILY becomes FAMILY'S, not FAMILIES.

The primary additions to the current BEAST's logical expression syntax are the three logical primitives defined by the syntax specifications

$$\text{(1)}\quad \text{IN}\quad \left\{\begin{matrix}\text{samplename} & \\ \text{segmntid} & \text{[boolprim]}\end{matrix}\right\}$$

$$\text{(2)}\quad \text{quantifier}\quad \left\{\begin{matrix}\text{inboolprim} & \\ \text{segmntid} & \text{[boolprim]}\end{matrix}\right\}$$

$$\text{(3)}\quad \left\{\begin{matrix}\text{ALL}\\ \text{EVERY}\end{matrix}\right\} \left\{\begin{matrix}\text{segmntid} & \text{boolprim}\\ \text{inboolprim} & \end{matrix}\right\}$$

TABLE I—Syntax for conditions on structures

```
samplestatement  :=  DEFINE SAMPLE name AS [refsegmntid] logexp
refsegmntid      :=  segmntid
segmntid         :=  segmntname | segmentname'S | segmntname (subscript)
logexp           :=  boolprim {AND} logexp | boolprim
                              {OR }
                     (WITH       )
                     (IN    WHICH)
                     (FOR   WHICH)
boolprim         :=  {FOR   WHOM } boolprim | NOT boolprim | (logexp) |
                     (HAS        )
                     (HAVE       )
                     (INCLUDE    )
                     (INCLUDES   )
    inboolprim | IN {sameplename           } | quantifier {boolprim['S]        } |
                    {segmntid [boolprim]   }              {segmntid[boolprim]  }
  {ALL  } {segmntid boolprim}
  {EVERY} {inboolprim       }
inboolprim       :=  (logexp) | logicalvar | numexp  relop  numexp
quantifier       :=  NO | ANY | A | AN | ONLY | [quantop]  integerexp
quantop          :=  EXACTLY | AT LEAST | AT MOST | MORE THAN | LESS THAN
where
logicalvar       :=  variable of type logical
numexp           :=  arithmetic expression
relop            :=  EQ | NE | GT | LT | GE | LE
integerexp       :=  expression with an integral value
```

1. It is often useful to test whether a segment is contained in another segment having certain characteristics, e.g., whether a PRODUCT segment is contained in a COMPANY segment of a particular sort or whether a segment with quarterly data is contained in a segment with particular annual data. To make such a test we have added a logical operator with the form

$$\text{IN}\ \left\{\begin{matrix}\text{samplename}\\ \text{segmntid [boolprim]}\end{matrix}\right\}$$

If a segment identifier immediately precedes the word IN then the test is applied to that segment. If no explicit identifier is used then the test is applied to the *reference segment* of the expression. For convenience let us call the segment being tested 'S'. Considering the form

S IN segmntid [boolprim]

the system first checks whether the segment containing S is of type segmntid. If no condition is specified on segmntid then the value of the IN phrase is the truth value of that inclusion test. If the segment S is contained in segmntid, then any condition on segmntid is also evaluated and the value of the IN phrase becomes the truth or falsity of the condition on the segment at the higher level.

If the IN operator has the form

S IN samplename

the test is TRUE if S is a member of the sample defined by *samplename* and FALSE otherwise. In this case the reference segment of the sample definition must be (1) the same as S or (2) a segment that contains S.

For example, using the SEO file, one might say

DEFINE SAMPLE S1 AS FAMILY'S IN HOUSEHOLD WITH AGEOFHEAD OVER 65

The reference segment is explicitly specified (after AS) as being the FAMILY segment. The segmntid is HOUSEHOLD, and the Boolean primitive modifying HOUSEHOLD is WITH AGEOFHEAD OVER 65. A particular FAMILY segment will be a member of the sample S1 if it is contained in a HOUSEHOLD segment with an elderly head.

For a second example, let us assume that there are two types of family segments, called COUNTRYFAM and CITYFAM, each of which may contain PERSON segments. We define as a sample called CITYFOLK all PERSON segments contained in CITYFAM segments by the statement:

DEFINE SAMPLE CITYFOLK AS PERSON'S IN CITYFAM'S

2. While the first logical operator gave us the ability to express conditions on the segments that contain the reference segment of an expression, the second operator puts conditions on segments that the reference segment may contain. This operator has the general form

$$\text{quantifier} \begin{Bmatrix} \text{inboolprim ['S]} \\ \text{segmntid [boolprim]} \end{Bmatrix}$$

As with IN, the segment S to which this phrase refers will be the reference segment unless it immediately follows a different segment identifier.

The quantifier operator tests whether S contains a specified number (given by the quantifier) of occurrences of some condition among its subsegments. The permissible forms for an quantifier are (1) A, AN, ANY, NO, ONLY (2) any of the relations

EXACTLY
AT LEAST
AT MOST
MORE THAN
LESS THAN

followed by an integral scalar expression or (3) simply an integer expression. A, AN, and ANY are equivalent to AT LEAST 1, and NO is equivalent to EXACTLY 0. The quantifier ONLY indicates no specific number of occurrences, but is TRUE if and only if S contains at that level only segments of type segmntid, and they satisfy the condition imposed on them, if any.

The condition referenced by the quantifier may be subsegments that satisfy some condition or simply the existence of the subsegments. A segment satisfying a condition is specified by either a segment identifier with a logical primitive, or simply an intrasegment boolean primitive (*inboolprim*) which is a condition made from attributes all in the same segment type. Since the value of an inboolprim is uniquely associated with a particular segment, an inboolprim is equivalent to a segment with a condition on it (See example below).

The following sample definitions illustrate the use of the quantifier logical operator applied to a file of household survey data.

DEFINE SAMPLE BIGFAMS AS FAMILY'S WITH AT LEAST 4 PERSON'S

This sample definition has FAMILY'S as its reference segment. The FAMILY segments in the sample are defined by a single logical primitive. According to the syntax specification, AT LEAST 4 is a *quantifier*, composed of a *quantop* (AT LEAST) followed by an *integerexp* which in this example is simply the number 4. In this example the quantifier is followed by a simple unqualified segment identifier, PERSON'S.

DEFINE SAMPLE CROWDED AS FAMILY'S IN HOUSEHOLD'S WITH AT LEAST 10 PERSON'S

The sample CROWDED is defined using both the primitives IN and a quantifier. IN is a condition on FAMILY'S because it follows immediately after the declaration of the reference segment. IN is followed here by the segment identifier HOUSEHOLD'S qualified by the phrase AT LEAST 10 PERSON'S. Evaluation of this expression involves a relatively complex computation on each enumeration unit, since for each FAMILY (a level 2 segment) it is necessary to find the total number of PERSON'S (at level 3) contained in the parent HOUSEHOLD segment at level 1.

```
DEFINE SAMPLE ELDERLYFAMS AS
    FAMILY'S WITH AT LEAST 2
    AGE'S > 60
```

This example shows one use of the construction called an intrasegment boolean primitive. Assuming that AGE is an attribute of the segment type PERSON, the quantifier phrase above would be equivalent to AT LEAST 2 PERSON'S WITH AGE > 60.

3. The final condition on a segment is also an operator applied to its subsegments. Though similar to ONLY, ALL and EVERY are evaluated using only the segment type indicated, and are independent of any other subsegment types which S may contain at the same level as *segmntid*. Also, a condition *must* be specified on the segment identifier. The words ALL and EVERY are equivalent. The general form is

$$\begin{Bmatrix}\text{ALL}\\ \text{EVERY}\end{Bmatrix}\begin{Bmatrix}\text{segmntid} & \text{boolprim}\\ \text{inboolprim} & \end{Bmatrix}$$

For example, considering a structure of the form

```
1 FAMILY
   2 CHILD
   2 ADULT [ASSETS]
```

we could say

```
FAMILY'S IN WHICH EVERY ADULT
    HAS ASSETS > 500
```

and the value of the expression would be independent of the contents of the CHILD segments contained in any FAMILY segment.

*Input and cutput*

An integral part of the BEAST language is its reliance upon *machine readable codebooks* for describing data files. The machine readable codebook includes a format description, including the physical and logical formats of the data file, the name and positions of all data items in the file, and the meaning of their permissible values. The BEAST system automatically references this information to interpret any user commands relating to a file.

As an example of a simple input request, suppose a user is investigating the relation between housing costs and income for different age-sex combinations. He knows that a given file, SURVEYFILE, contains the results of a sample survey useful to his investigati n; he also knows that the file contains at least the following four attributes of each respondent: age, sex, rent, and monthly income. In order to access this body of information using the BEAST, he writes:

```
SELECT SURVEYFILE
```

to designate SURVEYFILE as the current input file. The execution of the SELECT statement causes the BEAST to read the codebook associated with SURVEYFILE in preparation for an actual input request.

The codebook contains attribute names for each respondent item; suppose that those corresponding to the above attributes are AGE, SEX, RENT, and INCOME. For each attribute, the set of measurements for all respondents is represented as a column vector. To extract these attributes, the user writes in the BEAST:

```
GET SEX, AGE, INCOME, RENT
```

Execution of this GET statement causes four vectors to be extracted from the file and placed in working storage. There is no ordering rule for the input list; the order of the names has no relation to their physical arrangement on the file.

The remainder of this section shows how the "access by name" referencing of files can be extended to incorporate the more complex structures described in this paper. When only one segment type is considered there is no change from the current BEAST specification because there is no structural information. To signal the system that structural information exists in a file the user replaces the simple attribute list in a GET statement with either the name of an EU structure or an actual EU specification. Such a GET statement indicates that the structural information as well as the data values should be retrieved from the current input file. Similarly, an EU specification used in an output list will result in the display of only the attributes and structure indicated in the specification. As with the current BEAST, if no subscripts or sample qualifications are specified in an I/O list then every occurrence of the elements specified in the list will be retrieved or printed.

Using this form of I/O list one could write

```
DEFINE EU   DWELLING AS
  1 HOUSEHOLD [CITY, STATE]
  2 FAMILY [FAMTYPE]

SELECT SURVEYFILE

GET DWELLING (1. . .100)
```

The first statement defines DWELLING as a tree structure with two levels, the household level and the family level. There are two attributes at the household level; they give the city and state where the household is located. There is only one attribute at the family level, an indication of family type. The GET statement results in the extraction of the first 100 of these enumeration units from the data file called SURVEYFILE. The resulting number of FAMILY segments in these 100 HOUSEHOLDS is unknown, but it can be found by using the TOTAL function.

```
LET NFAMS = TOTAL FAMILY'S IN
HOUSEHOLD'S (1. . .100)
```

Since the segment identifier FAMILY'S is used with an explicit qualifier as the object of TOTAL, the value of the function will be a scalar equal to the number of FAMILY segments contained in the first 100 HOUSEHOLD segments.

### *A small BEAST program*

We conclude with a small but complete program utilizing the data structures and statement types described in this paper. This example also illustrates two other BEAST statement types, the REPEAT and COMPUTE statements. The iteration statement in BEAST is distinguished by the fact that its dummy argument is defined "by name" rather than "by value." This is a useful device permitting the dummy to be used on the left side of an assignment statement, to be only partially defined on entry of a repeat block, and to assume as a value any entity in the language that may be named. The general form of the iteration statement is given by

```
[label:] REPEAT FOR dummy 1 = namelist 1
         [AND FOR dummy 2 = namelist 2]. . .

         .
         .
         .
         END [label]
```

The dummy variable must be used in such a way within the range of a REPEAT that substitution of all elements of the list result in syntactically correct BEAST statements.

The COMPUTE statement is used to execute complex statistical procedures and print their results. The COMPUTE statement has the general form

```
COMPUTE procedure OF dataphrase [WITH
   optionsphrase]     [ON SAMPLE name]
```

The *procedure* may specify any of a number of procedures including cross-tabulation, correlation, multiple regression, and analysis of variance. The data to which the procedure is to be applied is specified in the *dataphrase*, and the exact form of the dataphrase depends on the procedure being invoked. The parameters of the procedure can be modified using the *optionsphrase*. One may, for example, specify that the residuals of a regression equation are to be printed as part of the output.

When the arguments of a procedure are at more than one level the number of "observations" derived from an enumeration unit equals the number of occurrences of the lowest level reference. In such a case the value of the higher level references are distributed over their subsegments giving a rectangular expansion of the tree structure. When the phrase ON SAMPLE *name* is appended to a COMPUTE statement the procedure is executed using only the observations that are included in the sample *name*. The reference segment of the sample must be at least as high as the lowest level attribute in the *dataphrase*.

Table II shows a program that uses the six subsets defined in the introduction as selection criteria for two cross tabulations using the Survey of Economic Opportunity file. The program will calculate and print a total of 12 cross tabulations, two on each of the six samples defined. Because the variables in the COMPUTE are at the PERSON level we may use either PERSON, FAMILY, or HOUSEHOLD level samples.

## CONCLUSION

Languages designed for statistics have tended to operate

TABLE II—Sample program

```
 #DEFINE THE STRUCTURE OF THE ENUMERATION UNIT#

 DEFINE EU DWELLING AS 1 HOUSEHOLD [SEXOFHEAD, AGEOFHEAD]
                           2 FAMILY [FAMTYP, REGION, URBANRURAL]
                              3 PERSON [WAGES, SALARY, AGE, SKILEVEL]

 DEFINE SAMPLE S1 AS FAMILY'S WITH NO AGE'S > 65

 DEFINE SAMPLE S2 AS HOUSEHOLD'S IN WHICH AT LEAST 2 PERSON'S HAVE
         (WAGES + SALARY) > 5000

 DEFINE SAMPLE S3 AS FAMILY'S WITH 2  AGE'S < 21

 DEFINE SAMPLE S4 AS PERSON'S IN HOUSEHOLD'S WITH SEXOFHEAD EQ 'F'

 DEFINE SAMPLE S5 AS FAMILY'S WITH REGION = 1

 DEFINE SAMPLE S6 AS PERSON'S IN FAMILY'S (WITH AT LEAST 5 PERSON'S
         AND WITH REGION EQ 7)

 SELECT SEO66  #SEO66 IS NOW THE INPUT FILE.#

 GET DWELLING (1...5 000)

L:REPEAT FOR  X = S1, S2, S3, S4, S5, S6  #ITERATE OVER SAMPLE DEFINITIONS#
      #THE PERMISSIBLE CATEGORIES FOR EACH VARIABLE ARE GIVEN IN THE
       CODEBOOK OF SEO66.#

 COMPUTE CROSSTAB OF SEX, AGE, SKILEVEL          ON SAMPLE X

 COMPUTE CROSSTAB OF URBANRURAL, SKILEVEL        ON SAMPLE X


 END  L     #REPEATS MAY BE NESTED TO ANY DEPTH.#

 STOP
```

upon the simplest data structures, while languages with facilities for the more complex structures have seldom had the statistical operations that have made the current version of BEAST attractive. By extending BEAST to include the tree structures described here we hope to increase the usefulness of the language without sacrificing any of the convenience of the current language. While the methods of referencing such structures have been stressed here it is nonetheless important to be able to manipulate such structures to add and delete individual segments and entire levels. We have not presented our tentative solutions to the problems of manipulating segments.

## REFERENCES

1 C W BACHMAN S B WILLIAMS
*A general purpose programming system for random access memories*
Proc FJCC Vol 26 1964 411-422

2 R M BALZER
*Dataless programming*
Proc FJCC Vol 31 1967 535-544

3 R E BLEIER
*Treating hierarchical data structures in the SDC time-shared data management system (TDMS)*
Proc 22nd Nat Conf Association for Computing Machinery 1967 41-49

4 R BUHLER
*P-STAT: An evolving user-oriented language for statistical analysis of social science data*
Princeton Computer Center Princeton Univ 1966

5 *Colingo project*
Colingo C-10 User's Manual Vol I II May 1968
Mitre Corp Bedford Mass AF 19 (628) - 5165

6 R W CONWAY et al
*CLP-The Cornell list processor*
CACM Vol 8 April 1965

7 W J DIXON editor
*BMD biomedical computer programs*
Univ Calif Press Berkeley Los Angeles 1967

8 Economic Growth Center
*Development of a generalized economic information retrieval system and data files*
Application for Nat Science Foundation Research Grant July 1966-June 1969 Principal Investigator Richard Ruggles Yale Univ 1966

9 H GELERNTER et al
*A fortran-compiled list processing language*
JACM Vol 7 April 1960

10 M GREENBERGER M JONES H JAMES JR D N NESS
*On-line computation and simulation: The OPS-3 system*
MIT Press 1965

11 M A GOETZ Chm
*The strategy of file organization*
Proc IFIP Congress 65 Vol 2 1965 May 24-29 460-479

12 Harvard University, Department of Social Relations
*The data-text system: A computer language for social science research designed for numerical analysis of data and content analysis text*
Preliminary Manual Harvard Univ 1967 Cambridge

13 I. B. M. Application Program
*Generalized information system application description* (GIS)
IBM Tech Pub Dept 1965 White Plains N Y

14 Inter-University Consortium for Political Research
*Machine readable codebooks and their use*
Inter-Univ Consortium for Political Research Nov 1967

15 R J JONES
*Data file two—A data storage and retrieval system*
Proc SJCC Vol 32 1968 171-181

16 A J PERLIS
*A definition of formula algol*
Carnegie Mellon Univ March 1966

17 N S PRYWES
*Executive and retrieval based extended machine*
Proc IFIP Congress 65 Vol 2 1965 May 24-29 460

18 J M SAKODA
*DYSTAL MANUAL: Dynamic storage allocation language in Fortran*
Brown Univ 1965 Dept of Sociology and Anthropology Unpublished manual

19 M SCHATZOFF
*Applications of time-shared computers in a statistics curriculum*
IBM Data Processing Division 1966 Cambridge Scientific Center

20 N R SINOWITZ
*Dataplus—A language for real time information retrieval from hierarchical data bases*
Proc SJCC Vol 32 1968 395-401

21 Social Systems Research Institute, Computation Division
*Socioeconomic Information Processing Service user's manual* (*SIPS*) preliminary corrections, February 10 1967
Univ of Wisconsin 1967 Madison

# Nanosecond threshold logic gates for $16 \times 16$ bit, 80 ns LSI multiplier

*by* LUTZ J. MICHEEL

*Air Force Avionics Laboratory*
Wright-Patterson Air Force Base, Ohio

## INTRODUCTION

Previous research and development efforts in digital monolithic integrated circuits and arrays were almost exclusively concerned with Boolean logic. However, by introducing threshold logic, considerable savings in gate count as well as in subsystem processing speed are evident. When logic subsystems, such as registers, adders, counters or combinational control logic, designed with common NOR logic, were replaced by subsystems employing threshold logic, average savings in gate count of three to one have been demonstrated.[15,16] Furthermore, the number of consecutive logic levels necessary to implement a given switching function, and thus the relative processing delay, is also generally reduced by the same ratio.

The full adder function requires two inverting threshold gates, and carry propagation is accomplished with only one gate delay per stage. Basic flip-flop types can be implemented by a single threshold gate. Advanced parallel adders for three addends would consist of three threshold gates per bit, and functional multipliers should also become practical in iterative array implementation.

For full utilization of the much greater logical capability of threshold gates, the employed technologies should be amenable to large scale integration which excludes hybrid approaches. Utilization of such monolithic threshold gates and arrays is possible in most kinds of computers, data processing and control equipment.

### *Threshold logic gates with nonlinear feedback*

Smith and Pohm have demonstrated the ultra-high speed capability of threshold logic gates in the form of RTL gates modified with negative, nonlinear current feedback.[1,2] In these gates $V_{CE}$ was clamped to $V_{ref} = V_{BE}$ by tunnel or backward diodes (Figure 1) which prevented both saturation and cutoff; thus, the transistor always operated in the ON condition near the $f_T$ peak. In reverse direction, of course, the diodes functioned as the familiar Baker clamp. Propagation delays between 5 and 1.2 ns were achieved in breadboard implementation with fan-out of 3 and 2. By varying the bias current $I_{bb}$, the authors implemented NAND, $\overline{\text{Majority}}$, and NOR for various values of fan-in $\leqq 8$ and also the threshold functions lying between these special cases. The gates were not amenable

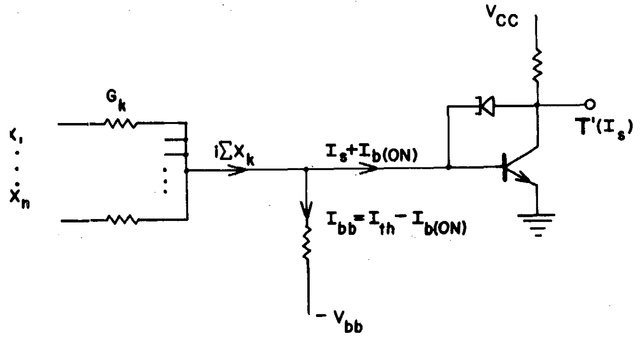


Figure 1—Modified RTL gate with tunnel diode feedback

to monolithic integration, however, because of the tunnel diodes and because high-accuracy resistors were required for current biasing.

### *Threshold logic gates for LSI*

When the first experimental Schottky barrier diodes became available,[3] circuits similar to the gates described in the first section, but having symmetrical transfer functions, were studied by the author. A pair of anti-parallel diodes were used for the collector base feedback (Figure 2). The plastic-encapsulated diodes, with molybdenum silicon interface, had 0.8 pF capacity at 0 V, and $V_D = 0.4V$ and $\Delta V/\Delta I = 25\Omega$ at 1 mA. Type 2N 918 transistors were selected for maximum $f_T$ and minimum $r_b'$. With 19.2 mW average power and i = 0.8 mA, the switching times $(t_d + t_r)$ and $(t_s + t_f)$ were between 1.65 and 1.85 ns.

Luce confirmed these switching time measurements,[4] and using experimental transistors SMX2-T with $f_T = 2.8$ GHz at $V_{CB} = 0$ V and $I_C = 2$ mA, he achieved propagation delays of 1.8 nswith only 5.6 mW circuit power. The 400 mV, TO-18 Schottky diodes contributed 2 pF Miller capacity. With reduced voltage swing of ±300 mV, Luce attained average $t_{pd} = 1.4$ ns and minimum $t_{pd} = 0.8$ ns.

Figure 3 depicts the symmetrical current-in, voltage-out transfer function and the summing point characteristics of the new gate which exhibits, at $I_{th}$ a switching step in $V_{BE}$ of only 23 mV.

A basic improvement of this symmetrical threshold gate over RTL circuits should be pointed out. In RTL it is the sum of input currents $\sum I_{in}$, plus the (negative) base bias current $I_{bb}$, which turns on the npn transistor. In the new threshold gate, the (positive) base bias current keeps the transistor at the threshold point $I_{th}$ in ON condition. The input current sum $\pm I_s, \sum I_{in}$ is merely required to switch the gate from $I_{th}$ to its high or low state.

First order worst case analysis of the basic 5-input

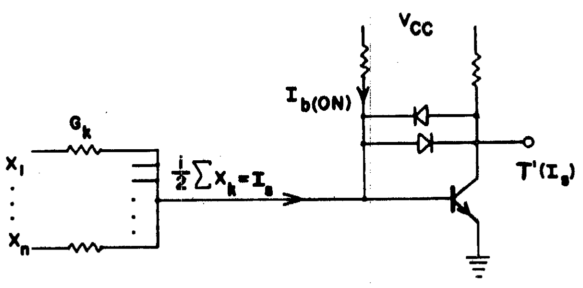


Figure 2—Threshold gate with Schottky diode feedback

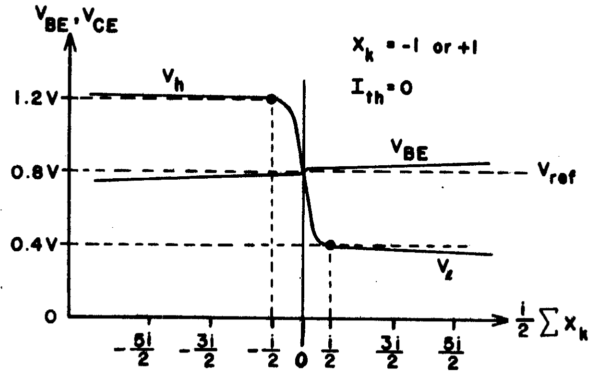


Figure 3—Transfer function of the gate shown in Figure 2

gate was performed under the assumption of temperature tracking of communicating circuits in monolithic LSI environment. The transistors should have $\beta \geq 40$ and $V_{BE}$ matching of ± 5 mV, while a resistor ratio tolerance of ± 3 percent is required. These characteristics can be attained in LSI with good manufacturing yield. The Schottky diodes should have $V_D = 0.4$ V ± 15 percent at 0.8 mA and very low capacitance for the high-speed version of the gate (i ≧ 0.8 mA). Several other versions are discussed in the next section.

Compatible metal-silicon Schottky diodes have been proposed (Mo[5,17]) and implemented (Mo,[6] Al[7,8,19] Pt[18]) as Baker clamps in integrated circuits which were mostly of the TTL type. The same technology is applicable to the modified RTL threshold gates. Transistors with $f_T \geq 2$GHz are now available for LSI utilization at low $I_C$ and $V_{CE}$ values.[9,10]

### *Various optimizations of the integrated gate*

While the experiments described in an earlier section were concentrated on high speed gates, with unity current step i = 0.8 mA, other circuit options would emphasize optimization in the following areas.

#### **Low power.**

For low fan-in gates, power consumption can be reduced by small input and collector currents and by lower collector voltage. The former implies transistor beta ≧80 in order to minimize the influence of absolute variations in $G_{bb}$, $G_k$ and $\beta$; unity currents $i \leq 0.2$ mA are attainable with this beta. The latter requires an active source or the collector current. This would improve the circuit dc performance since collector current variations between the high and the low output states would be minimized. Trade-off studies are required in order to determine whether current source or collector resistor contributes lower collector load capacities. The high area consumption and the low beta of lateral pnp transistors makes the current source less attractive for LSI circuits at this time.

**High input weight count.**

When many or all inputs are low, the high negative summation current must be accommodated by feedback diode current and collector resistor current. Minimum input current ($i/2 \leq 0.1$ mA) and high transistor beta are again required. Tantalum thin-film overlay resistors would provide high sheet resistivity for accommodating the large number of input resistors without requiring overly large substrate area consumption. Decoupling diodes[1,2] would ease the problems of leakage currents and of resistor tolerance requirements.

**Improved noise immunity.**

Two Schottky diodes in series per feedback branch (or simply two anti-parallel silicon diodes) would increase the voltage swing to $\pm 0.8$ V (or $\pm 0.6$ V). Collector biasing would be required in order to avoid saturation.[1,2]

**High fan-out.**

Collector biasing in combination with an emitter-follower output stage (Figure 4) would greatly improve the output drive capability[1].

**RC inputs.**

As Smith and Pohm pointed out, capacitive shunting of the input resistors would increase the gate switching speed for very low fan-out. Capacitive shunting, however, is an acceptable method only for low-noise array environment and for non-reversing switching transitions (no spikes), such as in a ripple carry.

**Reduced Miller effect.**

The detrimental Miller effect could be reduced if only one (symmetrical) Schottky diode were used with $V_D = \pm 0.4$ V at $I_D = \pm i/2$. Following a suggestion by Schuermeyer,[11] this type of diode can be obtained through very high concentration of surface states.

*Proposed functional LSI multiplier*

Recent advancements in the state-of-the-art of silicon processing for medium scale and large scale

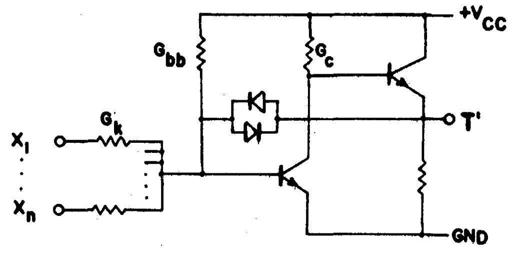


Figure 4—Collector biasing and emitter follower output

integration have made possible the implementation of monolithic arrays composed of the new threshold gate.

High densities with 11 mil$^2$ average area consumption per component have been achieved in pilot line LSI with good processing yield;[10] this includes intra/interconnections and three layers of metallization which facilitate optimum layout. The array was an 8-bit adder employing ECL trees with 1.2 ns carry propagation. The transistors have $0.15 \times 0.8$ mil emitters and 100 Ω/square base resistivity. The resistors were 0.5 mil wide with values in the 100 . . .400 Ω range and exhibited 6 percent ratio tolerance on 60 Ω /square material.

The threshold gates of the second section require clusters of equal resistors in the 1 . . . 4 kΩ range with 3 percent ratio tolerance. This tolerance could be attained with 0.5 mil wide resistors on 100 Ω/square base material. The 2 GHz transistors with $0.1 \times 0.4$ mil emitters discussed by Phillips *et al.*[10] should also be amenable to LSI in the early Seventies.

The 10 mW high speed gate with $i = 0.8$ mA uses 1 kΩ resistors; with fan-in of 5, this gate would encompass a substrate area of approximately $6 \times 13$ mil$^2$.

A $16 \times 16$ bit functional multiplier is proposed for LSI implementation using the 10 mW, 1 ns threshold gate. Figure 5 shows the matrix of multiplier cells in skewed form with all sum and output bits having a given binary weight arranged in the same column. Each cell $M_{ij}$ of the multiplier (Figure 6) is composed of a full adder and an AND gate which performs the multiplication. The cell in Figure 6 operates on $X_i'$ and $Y_j'$ and the adder inputs are $P_{ij} = X_iY_j$, $C_{(i-1)(j)}$, and $S_{(i+1)(j-1)}$. If the gates of the third section with symmetrical transfer function are used and if $T'_{w_o}(X_1,X_2, \ldots, X_k) = T'_{w_o}(I_s)$ is the inverting threshold function, all three multiplier-cell gates can be implemented with the threshold $w_o = 0$

$$P_{ij} = X_iY_j = T_{+0.5}(X_i, Y_j)$$

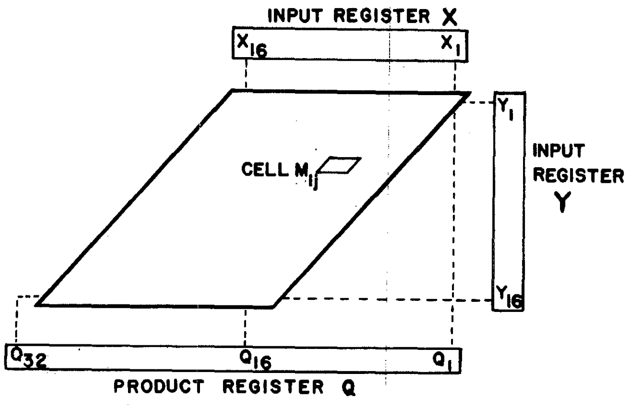


Figure 5—16 × 16 bit functional multiplier

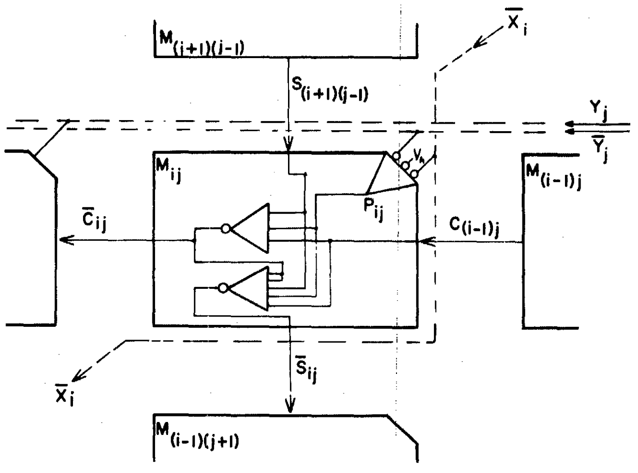


Figure 6—Functional multiplier cell

$$= T'_o(X'_i, Y'_j, +0.5)$$
$$C'_{ij} = T'_o (P_{ij}, C_{(i-1)(j)}, S_{(i+1)(j-1)})$$
$$S'_{ij} = T'_o (P_{ij}, C_{(i-1)(j)}, S_{(i+1)(j-1)}, 2C'_{ij}) .$$

The proposed multiplier would be implemented on four LSI dies with 64 iterative cells each (Figure 7) with two layers of metallization. For attaining optimum layout and, thereby, higher component density, an implementation with three layers may be preferable.[10] The three gates per cell would occupy an area of 13 × 18 mil$^2$, and each LSI die would have an area of ≈ 115 × 155 mil$^2$. For 10 mW power per gate, the array will consume 1.92 W power, and 44-pin 1 × 1 in$^2$ stud packages would provide adequate thermal management.

The multiplier cells are used in two complementary ways—as Type 1 cells with positive inputs/negative

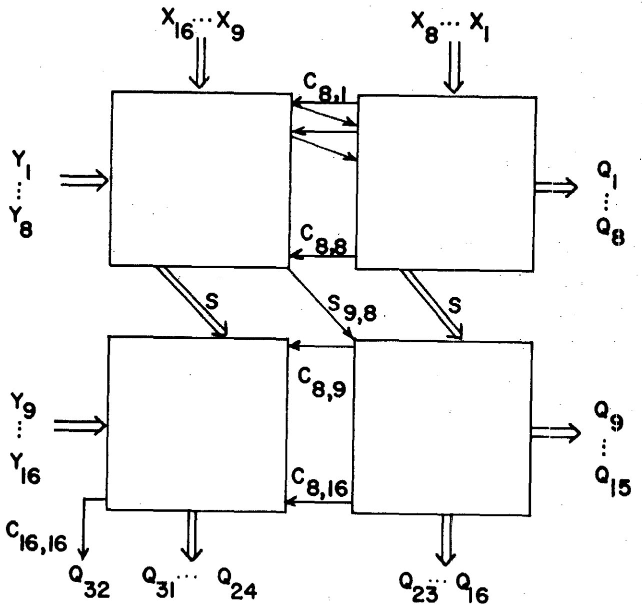


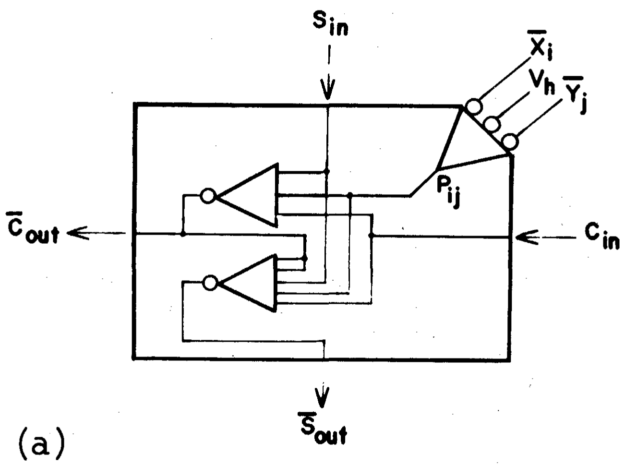
Figure 7—16 × 16 bit functional multiplier on four LSI dies

outputs (Figure 8a) and as Type 2 cells with negative inputs/positive outputs (Figure 8b). Equivalent to the odd/even levels in NOR logic design,[12] alternating matrix columns (Figure 9a, b) are composed of Type 1 cells using inverted inputs $X'_i$ (i = odd, e.g.) and of Type 2 cells using true inputs $X_{i+1}$ (i + 1 = even). Only one bus connection to the matrix is required per flip-flop in the X-register, whereas both $Y_j$ and $Y'_j$ are bussed through each horizontal row. An additional column i = 17 of carry inverters (Figure 10) converts $C_{(16)j}$ into $S_{(17)j}$.

Although the average carry ripple length is much shorter[13] than the full length of each partial product adder (having j = const.), no carry look-ahead circuitry is included since it would corrupt the iterative structure of the multiplier and also the approach of minimum wafer area consumption of the LSI array.

The worst case multiplication time $t_M$ for two k-bit numbers includes 2k − 1 carry delays and k − 1 sum delays (Figure 10). Three nanoseconds should be allowed for each package-to-package transition $t_{tr}$ assuming matched transmission lines, and a setting time of less than 2 ns is required for the output flip-flops Q each of which consists of a single threshold gate.[14] For k = 16,

$$t_M = t_{pd}(\text{AND}) + 31\ t_{pd}\ (\text{Carry}) + 15\ t_{pd}\ (\text{Sum}) + 5\ t_{tr} + t_{set}$$



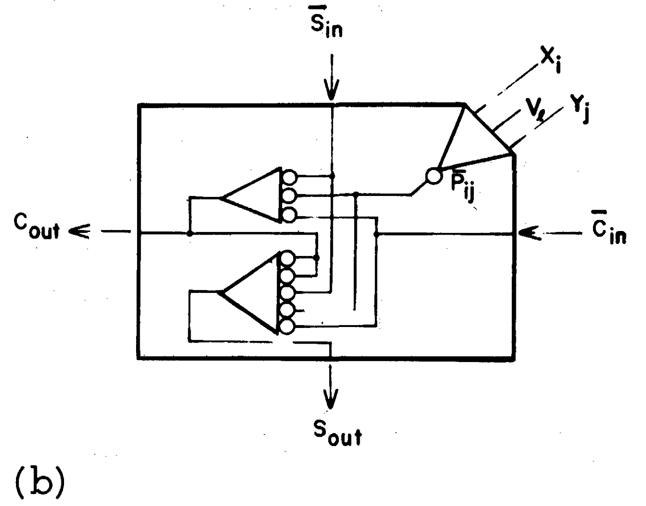


Figure 8a—Type 1 cell

Figure 8b—Type 2 cell

$$= (1 + 31 + 30 + 15 + 2) \text{ ns}$$

$$= 79 \text{ ns.}$$

## CONCLUSIONS

New low-power nanosecond threshold logic gates which are amenable to monolithic LSI have been discussed. These gates require high-performance integrated devices, and the necessary advanced silicon processing techniques should be available with high manufacturing yield in the early Seventies. Functional LSI multipliers with 80 ns multiplication time for two 16-bit numbers have been proposed. Such multipliers and

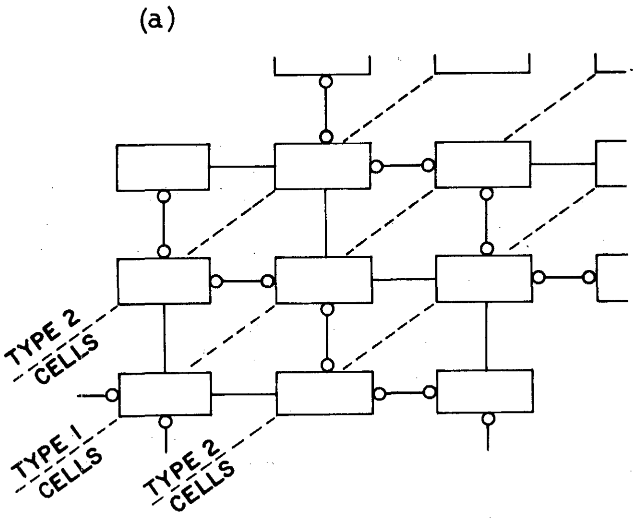


Figure 9a—Alternating type 1/type 2 cells in the multiplier matrix

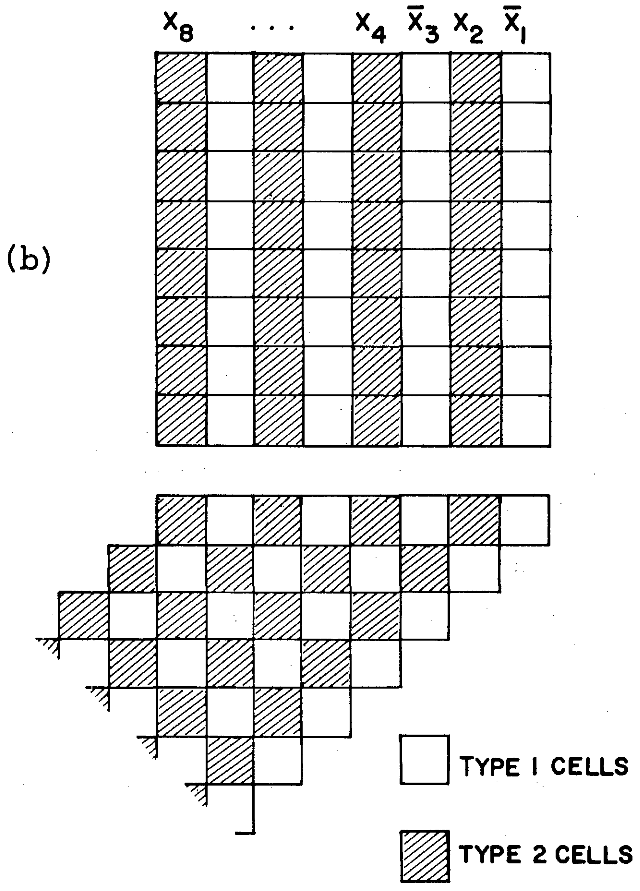


Figure 9b—Alternating columns X/$\bar{X}$ composed of type 1/type 2 cells

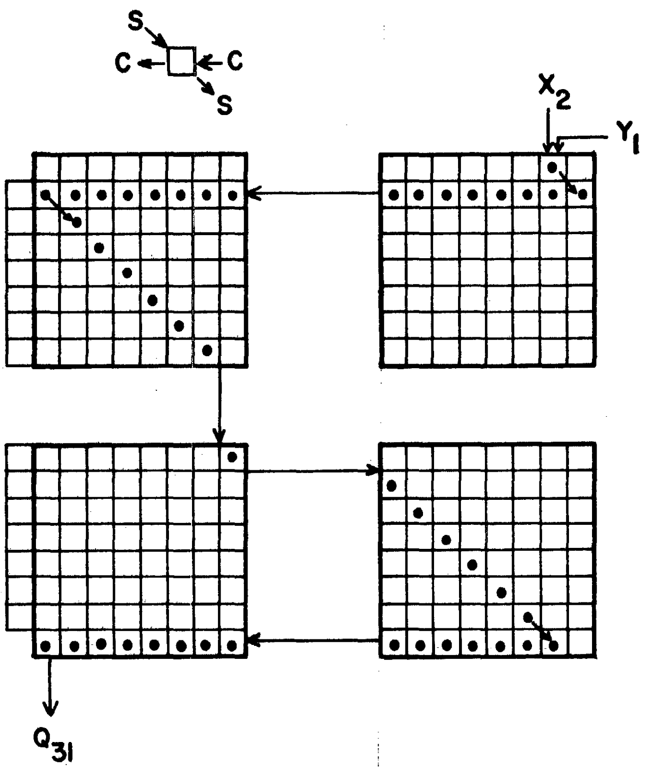


Figure 10—Longest path for worst-case multiplication time

similar monolithic LSI arrays, e.g., high-speed adders, counters, and control logic subsystems, can be advantageously implemented with threshold logic;[15] the average savings in gate count is 3:1, and the number of interconnections is reduced by 2:1 or more. LSI arrays with the new 10 mW, 1 ns threshold gate would be applicable to future ultra-fast, low-power data processing systems. Practical procedures for logic design with threshold logic gates were published elsewhere by Winder.[16]

## ACKNOWLEDGMENTS

The stimulating discussions with R. Winder, R. Luce, J. Tellier, and C. Huang are gratefully acknowledged.

## REFERENCES

1 W R SMITH A V POHM
*A new approach to resistor-transistor-tunnel diode nanosecond logic*
IRE Trans EC Vol 11 Oct 1962 658-664

2 W R SMITH
*Resistor-transistor-backward diode nanosecond logic*
Semiconductor Products and Solid-State Tech Vol 6 March 1963 17-23

3 Samples of hot carrier diodes developed under Contract AF33(615)-2629 by Motorola Inc Semiconductor Div for the U. S. Air Force Avionics Lab
The samples became available in Oct 1965.

4 R L LUCE
Personal communication May 1966

5 W. SEELBACH
Monthly Status Letter for May 1966 Motorola Inc under Contract AF33(615)-5205 with the Air Force Avionics Lab
See also Reference 10

6 Y TARUI Y HAYASHI H TESHIMA T SEKIGAWA
*Transistor Schottky-diode integrated-logic circuit*
Internat Solid-State Circuits Conf Phila Pa Feb 1968

7 R A ALDRICH
*Low storage Schottky barrier diode transistors*
Internat Electron Devices Meeting Wash D C Oct 1968

8 J E PRICE
*A high-speed integrated Schottky-diode transistor logic circuit*
Internat Electron Devices Meeting Wash D C Oct 1968

9 W SEELBACH D METZ
*Compatible semiconductor thin film techniques*
AF Avionics Lab Tech Rpt AFAL-TR-66-305 Oct 1966
AD 802 677 Prepared under Contract AF33(615)-2629 by Motorola Inc Semiconductor Products Div Phoenix Ariz

10 C PHILLIPS G RUPPRECHT F LEE
*Advanced integration techniques for low power, 100-200 MHz digital processing*
AF Avionics Lab Tech Rpt AFAL-TR 68-226 Sept 1968
AD 843 997 Prepared under Contract AF33(615)-5205 by Motorola Inc Semiconductor Products Div Phoenix Ariz

11 F SCHUERMEYER
Personal communication Sept 1968

12 G A MALEY J EARLE
The Logic Design of Transistor Digital Computers
Prentice-Hall Inc 1963 Englewood Cliffs N J

13 B GILCHRIST J H POMERENE S Y WONG
*Fast carry logic for digital computers*
IRE Trans EC Vol 4 Dec 1955 133-135

14 J I AMODEI D HAMPEL T R MAYHEW R O WINDER
*An integrated threshold gate*
Internat Solid-State Circuits Conf Phila Pa Feb 1967

15 R O WINDER
*The status of threshold logic*
First Annual Princeton Conf on Info Sciences and Systems Princeton N J March 1967

16 J H BEINART D HAMPEL K. PROST R O WINDER
*Integrated threshold logic for LSI*
U S AF Avionics Lab Final Rpt No AFAL-TR-69-195 on Contract F33615-68-C-1536 Prepared by RCA Advanced Communications Lab Somerville N J
Published in Aug 1969 AD 857 477

# Silicon-on-sapphire complementary MOS circuits for high speed associative memory*

*by* J. R. BURNS and J. H. SCOTT

*RCA Laboratories*
Princeton, New Jersey

## INTRODUCTION

The utility of associative memory in a wide variety of information handling systems has been long recognized and in the early 1950's such memory systems were proposed for implementation through cryotron logic and storage arrays. Cryogenic element technology afforded the ingredient of compatible logic and memory within a basic cell, a requirement essential to the practical realization of associative memories. To date, such an approach has not been successful due mainly to processing difficulties connected with thin film elements operating in a liquid helium environment. Other approaches, involving the use of multi-apertured magnetic elements, have been proposed and implemented, but the resultant cost was prohibitive due to complexities of peripheral electronics as well as the magnetic storage element itself. Furthermore, systems of this type have relatively long parallel search times ($\sim$ 10 $\mu$secs) especially if access is on a serial-by-bit basis. These considerations have seriously limited the applicability of associative concepts in all forms of data processing and have resulted in a situation where system designers do not consider associative memory as a solution to a given problem in spite of many obvious advantages in applications such as sorting, merging, pattern recognition, and most recently, memory allocation in time shared computers.

Many of the objections mentioned above are not valid today because of the rapid evolution of integrated circuit technology. This is particularly the case for semiconductor memory arrays where the universality and regularity of such sub systems take full advantage of the low cost potential of Large Scale Integration (LSI). Considerable effort has been expended throughout the industry on high speed random access memory arrays having non-destructive read-out in the sub-100 nanosecond range where the cost of competitive magnetic memories is dictated by the high quality peripheral electronic circuitry. Although a substantial part of this effort has been on bipolar memories, the dominant trend is toward MOS memories because of the simpler processing technology, lower power dissipation, and smaller silicon area per bit, all of which lead to lower cost systems. Monolithic silicon MOS memories generally take two forms, i.e., single polarity MOS arrays, usually P type, and complementary MOS, a unique circuit configuration capable of higher speed and extremely low power but at the expense of more complex processing technology and slightly higher costs. This is the approach taken here for the realization of sophisticated associative memory with one important difference, namely, the utilization of thin film silicon-on-sapphire technology[1] for the fabrication of high quality complementary MOS arrays. Silicon-on-sapphire combines the best features of monolithic

* The research reported in this paper was sponsored by the Electronic Research Directorate, Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, New York under contract F30602-68-C-0197.

silicon and thin film integrated circuits through the epitaxial growth of thin films of single crystal silicon-on-sapphire substrates which can be selectively removed so that all parasitic reactance which degrades the performance of monolithic circuits is effectively eliminated. Coupled with the improved circuit performance is a potentially simpler processing sequence for CMOS integrated arrays (requiring only two noncritical source-drain diffusions) which will substantially reduce costs as well.

*Associative memory design*

## General considerations

Several considerations influence the design of an associative memory array, the majority of these having to do with the sophistication required of the memory. Based on requirements believed to be minimal in most associative applications, the following features are desirable:

1. Normal operation as a read-write random access memory (having high speed non-destructive read out) in addition to completely parallel associative search operation.
2. "Masked" search capability so that any part of the total field can be eliminated from the search word. This will also provide a "masked" write wherein partial updating of the field of a selected word is possible.
3. Modular array design so that associative memories of arbitrary numbers of words and bits per word can be constructed by "wired OR" connections of the word and bit lines of individual modules.

Accordingly, the module was chosen to be one of four words each four bits long and has the basic block diagram shown in Figure 1.

Operation of the module is summarized in the following Table I.

As shown, the module performs as a read-write memory in addition to the ability to perform a completely parallel search. In the "don't care" condition where both of the bit line pairs are "0," any of the digits to be completely masked off in this condition will not produce a mismatch signal in any word regardless of the contents of that bit in the word. Design of the basic cell which performs these various functions is discussed in the next section.

## Associative cell design

The circuit diagram of the basic cell, designed to implement the aforementioned functions, is shown in Figure 2 and is seen to consist of 14 MOS devices, 10 N channel and 4 P channel. The flip-flop portion consists of the cross-coupled CMOS inverters $N_1$, $P_1$, and $N_2$, $P_2$. To write a "1" into the cell, W and $D_1$ are r ised to $+V_0$ volts and $D_2$ remains at ground. This combination cuts off $P_3$ while the series combination of $N_3$ and $N_5$ pulls the "0" side of the flip-flop down toward

TABLE I—Associative module system operation

| FUNCTION | WORD AND BIT LINE CONDITIONS | | RESPONSE |
|---|---|---|---|
| Write | Wi = "1"; $D_{1j}$ = "1," $D_{2j}$ = "0" | | Write "1" in j<sup>th</sup> bit of i<sup>th</sup> word. |
| | Wi = "1," $D_{1j}$ = "0," $D_{2j}$ = "1" | | Write "0" in jth bit of ith word. |
| Read | Wi = "1" All D lines = "0" | | Non-destructive read of ith word—stored contents determined by presence or absence of $I_s$ on lines $D_{2j}$. |
| Paralle Search | All Wi = "0" | | |
| | $D_{1j}$ = "1" | $D_{2j}$ = "0" | Search for "1" in jth bit. |
| | $D_{1j}$ = "0" | $D_{2j}$ = "1" | Search for "0" in jth bit. |
| | $D_{1j}$ = "0" | $D_{2j}$ = "0" | Don't care. |
| | | | Mismatch of any bit in word indicated by current on W lines. |

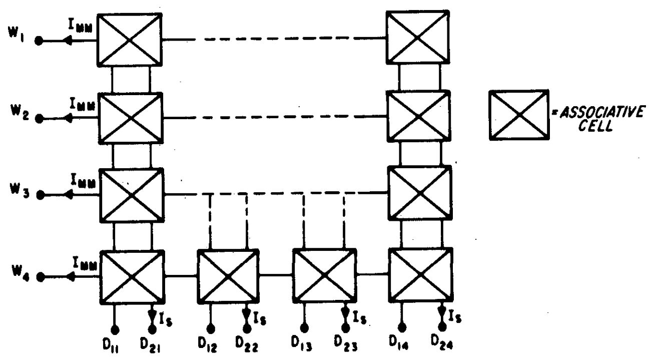


Figure 1—Associative memory block diagram

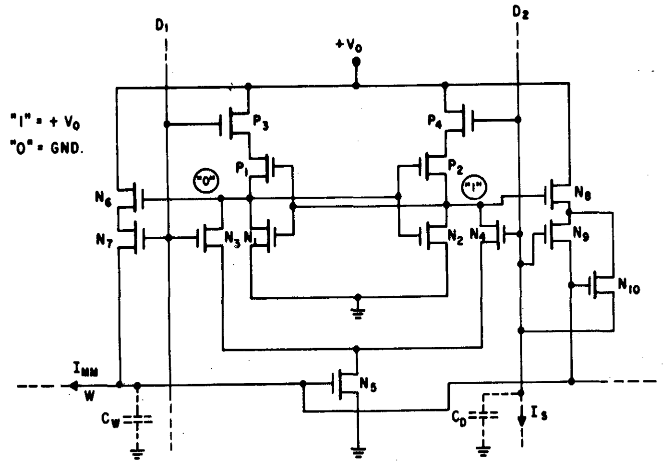


Figure 2—Complementary MOS associative cell

ground and after one stage delay the "1" side is up at $+ V_0$. Similarly, a "0" is written by raising W and $D_2$ to $+ V_0$ with $D_1$, at ground potential. Note that when both lines are grounded, and W is high, the state of the cell is unchanged.

Non-destructive read out is obtained by again selecting W, thereby turning on transistor $N_{10}$, and keeping all D lines at ground. Depending on the state of the cell, $N_8$ is either on or off and a large or negligible small current is produced on the low impedance $D_2$ line.

Mismatch detection by means of a parallel search is accomplished with all W lines grounded and placing the search criterion on each bit line pair, i.e., $D_1 = + V_0$, $D_2 = 0$ for "1"; $D_1 = 0$, $D_2 = + V_0$ for "0" and $D_1 = D_2 = 0$ for "don't care" or "Ø". Transistors $N_6$, $N_7$, $N_8$, and $N_9$ form the local exclusive OR circuit. If the stored information mismatches the information on the bit lines, one of the pair of $N_6$-$N_7$ or $N_8$-$N_9$ will form a conducting path from the positive supply to the W line (at ground potential) and produce a large current (~1 mA) in the W line. Both pairs will be cut off if there is a match or if a "don't care" condition prevails in that bit location. Since all such circuits are OR'd together across the word, a match occurs only if all exclusive OR gates are open or a negligible small current appears on the W line. Any bit of the word mismatching the search bit will generate a mismatch current for the entire word.

It should be noted that read out and mismatch detection are both accomplished by current sensing in a low impedance line. This is an extremely high speed operation as the relatively large capacitance on the word and digit lines can be swamped by the low input resistance of a grounded base bipolar and the voltage conversion done at the relatively low capacitance collector and at essentially the same current level. (A complementary emitter follower performs more than adequately as a combination drive-sense circuit on both word and digit lines.) In high speed table look-up applications, such as "paging" in time shared computers, fast parallel search and access is extremely desirable as this function is carried out once every main memory cycle.

Another aspect of current sensing on the mismatch-line is that the magnitude of the mismatch current is directly proportional to the number of bits in error in that particular word. Utilization of analog detection circuitry on this line will then enable the determination of the word which most closely matches the search word, independent of the significance of the bit. The so-called "proximity match" condition is quite useful in many aspects of pattern recognition, for example, or other applications where incomplete information is available for the search criterion.

*Processing of silicon-on-sapphire COS/MOS*

### Technical considerations

Great difficulty has been experienced and reported by workers[2] attempting to build high quality, active silicon devices on sapphire substrates by the straightforward application of standard bulk silicon technology to these heteroepitaxial films. These difficulties can be traced, in general, to two problems.

The first is contamination from the substrate, epitaxial system or handling procedures, and the second is due to disorder in the epitaxial layer caused by the growth interface. It can, therefore, be expected that devices and circuits fabricated in heteroepitaxial material must have the silicon processing adjusted in order to account for these deviations in properties.

TABLE II—Physical characteristics of heteroepitaxial system components

| | Silicon Si | Sapphire $Al_2{}_3O$ |
|---|---|---|
| Crystal Unit cell (Å) | face-centered cubic a = 5.4301 | r = 4.758 a = 12.991 |
| Density (g/cc) | 2.33 | 3.98 |
| Hardness (Mohs) | 7 | 9 |
| Melting point (°C) | 1412 | 2030 |
| Dielectric constant | 11.7 (500 Hz − 30 MHz) | 9.4 (1 to C) (100 Hz − 100 kHz) |
| Dissipation factor tan $\delta$ | $10^{-3} - 10^{-4}$ | $10^{-3} - 10^{-4}$ |
| Refractive index | 3.4975 at 1.357 $\mu$ | 1.7707 at 5461 Å |
| Thermal conductivity cal/cm sec·°C at 25 °C | 0.30 | 0.065 (60° to C) |
| Thermal expansion coefficient 1/ °C(25 = 800°C) | $3159 \times 10^{-6}$ | $8.4 \times 10^{-6}$ |

Table II is a comparison of some of the physical characteristics of the components of the heteroepitaxial system that must be taken into account if high quality silicon-on-sapphire devices are to be built. From this data, it is evident that some physical stress and disorder due to the mismatch of these characteristics is inevitable.

The effects of disorder and strain on the properties of bulk silicon are well known, e.g., base "push out" in bipolar transistors. Comparison of what is known to occur in bulk and what is observed in SOS yields some insight into the processing considerations. The most severe problems are:

1. Accelerated Diffusion
2. Accelerated Oxidation
3. Contamination

The change of diffusion coefficient in bulk silicon is a function of surface concentration and dislocation density. The distribution of disorder sites in SOS has been shown to be highest at the silicon to substrate interface and decreases as the thickness of the film increases.[3] Due to this distribution, there is a change in diffusion coefficient causing the impurities to move faster as they penetrate the film and therefore aterall diffusion can increase with depth. The resultant diffusion profiles are depicted in Figure 3. The following Figure 4 is a photograph of an actual "angle lap and stain" demonstrating the results of too long a diffusion of the source and drain regions. Note the resulting short of the source to the drain is at the silicon to sapphire interface. Because SOS has no bulk to dilute the fast diffusing contaminants plus the additional complication that the substrate can contribute to the contamination (A1, $O_2$, etc.) much greater care must be taken in handling and substrate preparation. This

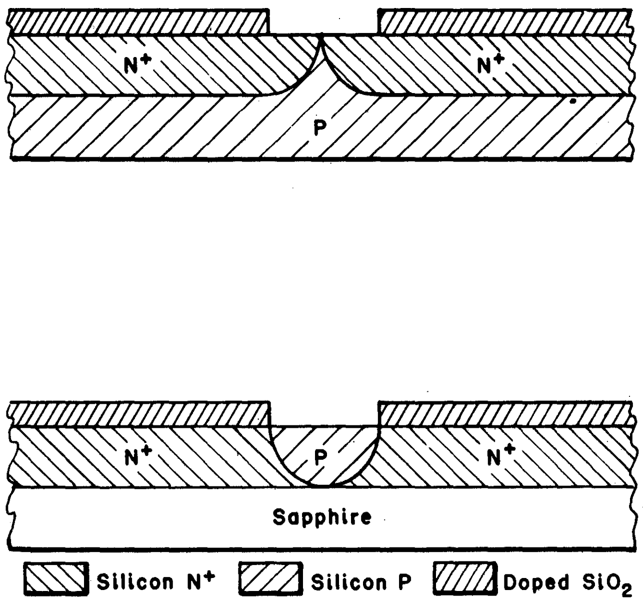


Figure 3—Diffusion failures in bulk silicon and silicon-on-sapphire

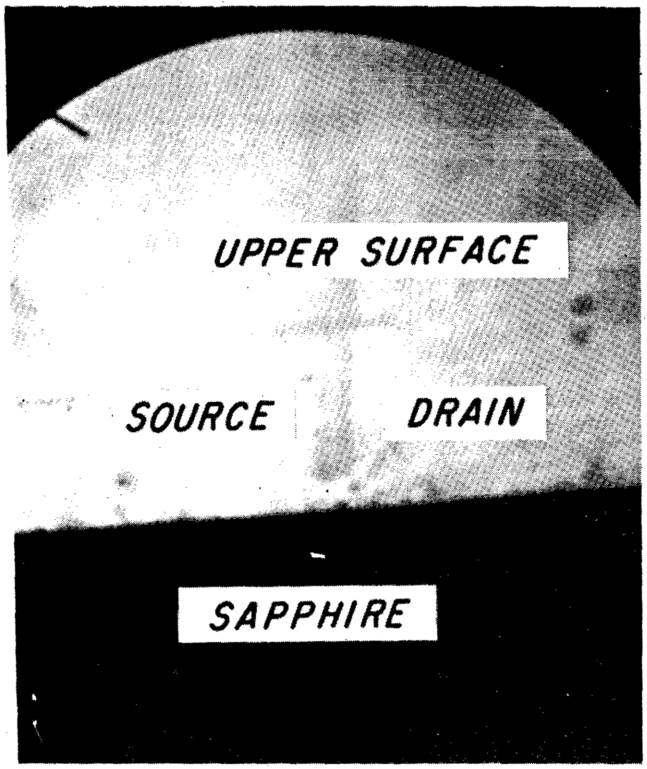


Figure 4—Photomicrograph of diffusion failure in SOS

consideration is further compounded by the affinity of contaminants for disorder sites.

Finally, oxidation and its effects must be considered in the light of what is known to occur in bulk silicon, for this is the pillar on which silicon technology is built. Here, one finds three major effects. The first is dissolution of $0_2$ from the ambient and the generation of donor states with reported values on the order of $10^{18}$/cc. The second is the precipitation of these impurities causing large changes in mobility and, finally, segregation of impurities in the oxide.[4]

From the previous discussion, it is obvious, without going into the details of the phenomenon involved, that bulk silicon technology is not directly applicable to the fabrication of high quality complementary MOS devices on insulating substrates. The necessary alterations in the process involve elimination of oxidation where possible and minimization of the time that the wafer is exposed to high temperatures. In addition, advantage can be taken of the thin film nature of the technology by utilizing the "deep depletion" MOS structure,[5] thereby enabling construction of complementary devices in a film of common conductivity type.

Figure 5(a) depicts a wafer of silicon-on-sapphire with a 300°C deposited oxide defined by photolithographic techniques in the pattern to etch away that silicon not utilized by active devices. Figure 5(b) shows the pattern left after the silicon is etched from the undesired region.

After the desired pattern is achieved in the silicon, thin layer of boron doped oxide [cross-tracked area Figure 5(c)] covered with pure $SiO^4$ is deposited (300°C) and etched into the desired $P+$ regions. This is followed by phosphorus doped oxide covered by pure $SiO^4$ and etched to define the $N+$ regions as shown in Figure 5(d).

This structure has never been above 300° C and has the appropriate doped oxide source defined in the $P+$ and $N+$ regions with the channel regions clean and free of oxide. The wafer is then subjected to its only high temperature treatment for the time required to drive in the diffusants and grow the channel oxide. This is indicated in Figure 5(e) with the appropriate diffusions drawn in. The final device structure (see Figure 6), including metallization, shows the built-up oxides at the edge of the active gate region minimizing the parasitic overlap capacitance.

The metallization utilized to complete these structures was evaporated aluminum and posed some problems in continuity due to the relatively large silicon steps the metal was required to pass over ($\approx$ 1 micron). Figure 7 is a scanning electron photograph of one such crossover. Note that the metal is thinner than the one micron of silicon and that the continuity appears sus-

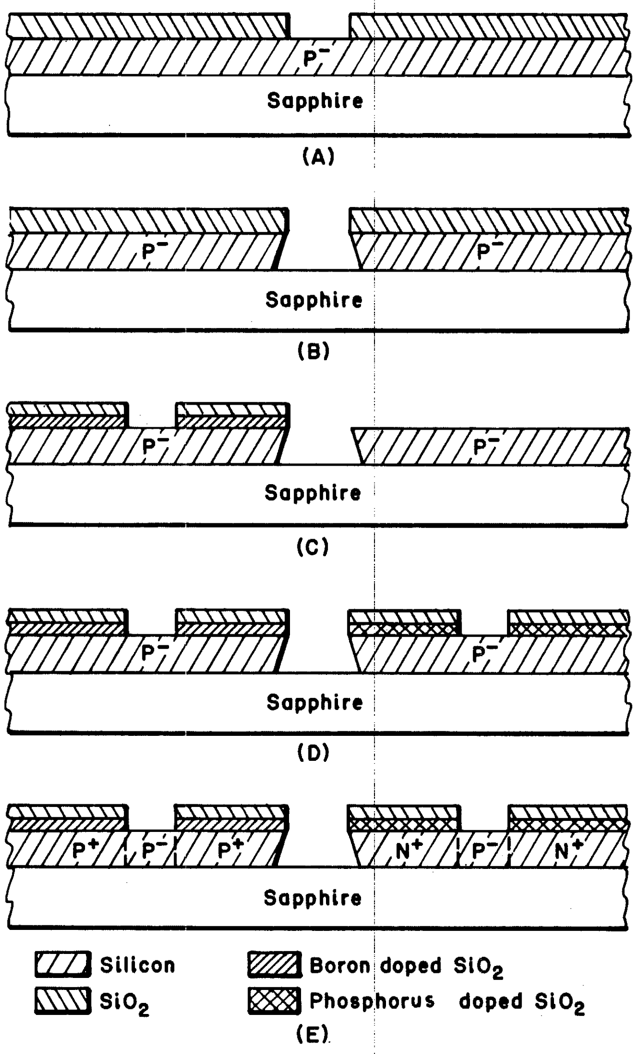


Figure 5—Low temperature CMOS Process

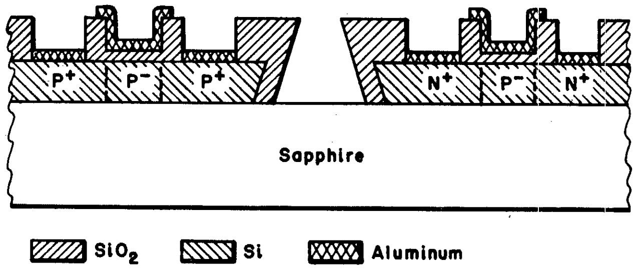


Figure 6—Final device structure

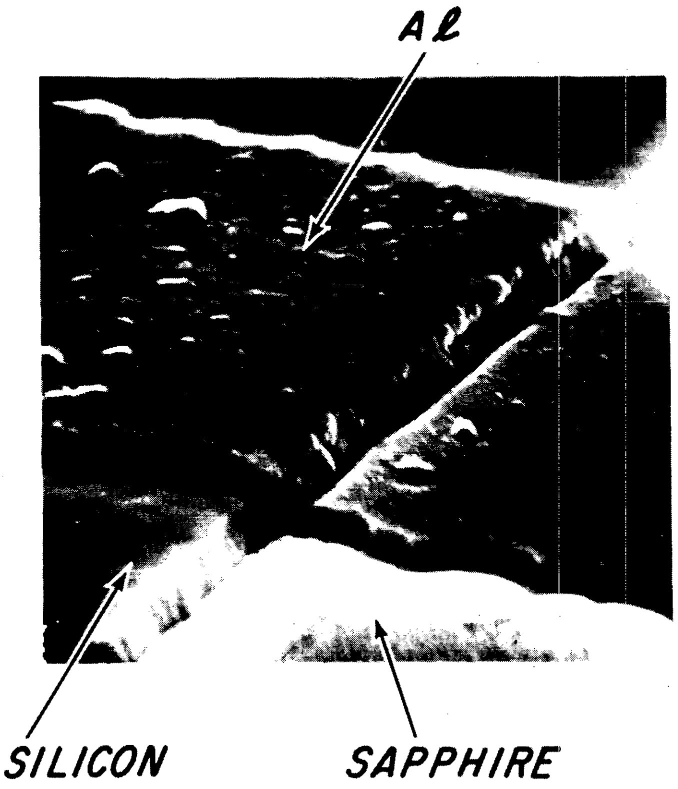


Figure 7—Aluminum metallization over oxidized silicon edge

pect. In fact, it was continuous. By increasing the thickness of the aluminum used to 15,000 Å or 1.5 microns, this problem was virtually eliminated.

## *Unique features of SOS technology*

Several significant advantages result from the utilization of SOS in the areas of process simplification as well as improved device and circuit performance. The use of the "deep depletion" MOS[5] eliminates the need for a difficult counter diffusion to form complementary devices while selective silicon removal restricts the critical silicon areas to the channel regions of the transistors since all metal interconnects are routed over the sapphire substrate. This gives complete freedom from metal to substrate shorts and spurious channel formation, two major sources of yield reduction in monolithic MOS arrays.

The most obvious advantage of this technology is the substantial improvement in circuit speed due mainly to the virtual elimination of all parasitic capacitances within the array. As shown in Figure 6, the through diffusion of the source and drain contacts to the sapphire reduces the contribution of the junction capacitance to the side-wall area, one dimension of which is only 1 micron thereby cutting this capacitance by approximately two orders of magnitude over a bulk device of the same surface dimensions.

Doped oxides as solid diffusion sources serve to further reduce parasitics in the form of gate overlap and crossover capacitances and all wiring capacitance is completely eliminated. Combined with the low threshold voltages (typically 0.5 volt enhancement for both device types) and the high field effect carrier mobilities of the transistors, the overall result is the realization of the full high frequency capability of MOS devices within an array environment. Inasmuch as the gain bandwidth product of the MOS is comparable to that of double diffused bipolar devices, circuit speeds approaching those obtained with non-saturating bipolar logic gates (nano-second stage delays) can indeed be achieved with this technology while retaining the power and noise immunity features inherent in complementary MOS circuitry.

*Integrated circuit design and experimental results*

The fabrication of the associative array requires a total of five photo-masks each of which was generated with the aid of an automatic drafting machine. These masks define, in order of processing sequence, the isolated silicon islands, boron doped oxide, phosphorous doped oxide, contact opening and aluminum metalization patterns. Heavily doped N+ silicon bars are used throughout as a first layer of interconnection. Extension use of symmetry and mirror imaging was used in the layout as an effective means of reducing chip area. A photomicrograph of the completed silicon-on-sapphire associative array is seen in Figure 8 with the bonding pads appropriately labeled. The chip has an active area of 77 × 53 mils, is packaged in a 14 lead flat pack, and contains a total of 224 MOS devices. A test complementary inverter is included within the patern for initial wafer evaluation. Each transistor in the array (including the test units) has identical channel widths of two mils, lengths of 0.4 mils, and channel oxide thickness of 1800 Å. Characteristics of typical test devices are shown in Figure 9. Based on these parameters and the assumed lateral diffusion of about 1 micron on both the source and drain regions, field effect mobilities of 150 cm$^2$/volt second for holes and 300 cm$^2$/volt second for electrons are obtained from the characteristics.

Experiments conducted on fully packaged arrays show that a storage cell can be written into with a 10 volt, 10 nano-second duration pulse with the array at the 10 volt supply level. Minimum sense current during read out is 1 mA as is the minimum

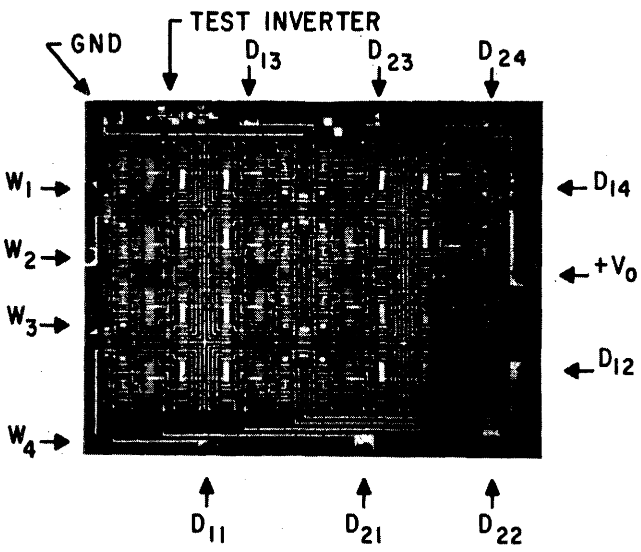
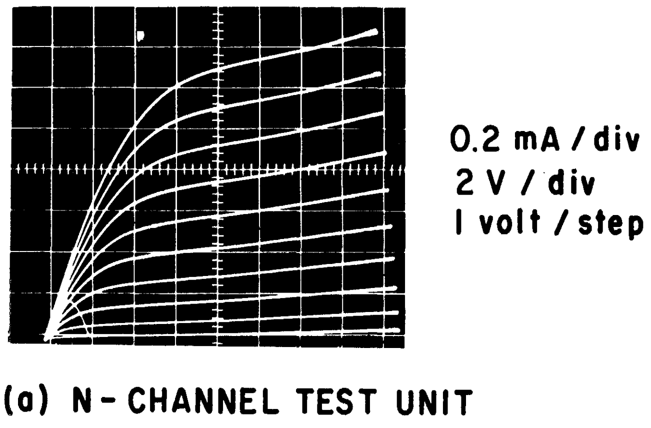


Figure 8—16 bit SOS associative array



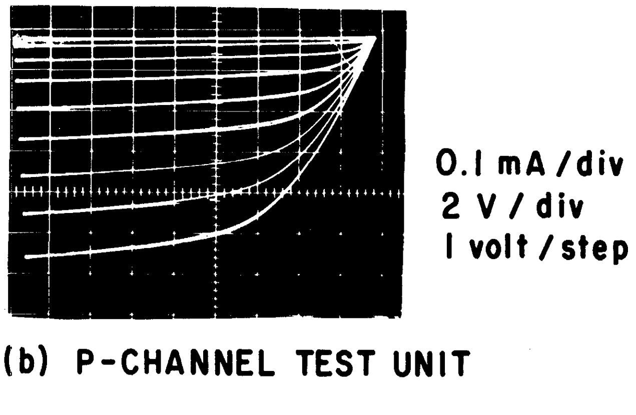
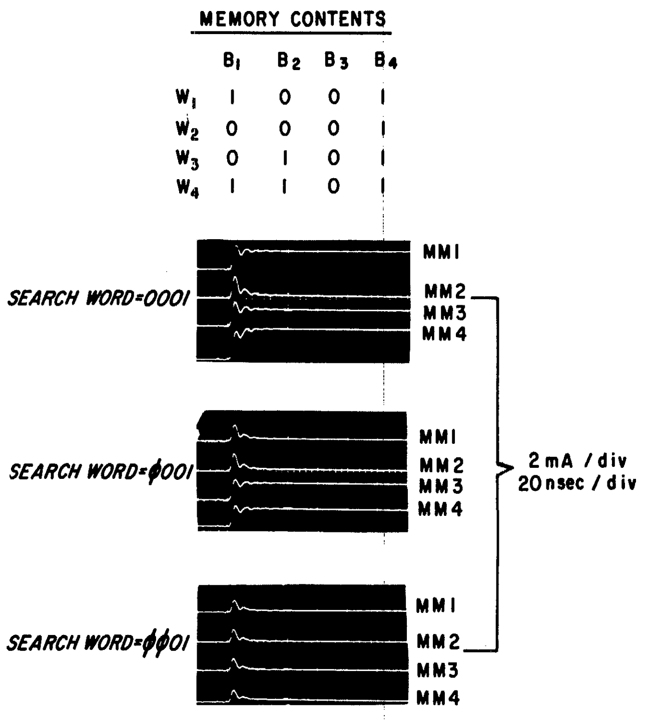


Figure 9—Test device characteristics



| | $B_1$ | $B_2$ | $B_3$ | $B_4$ |
|---|---|---|---|---|
| $W_1$ | 1 | 0 | 0 | 1 |
| $W_2$ | 0 | 0 | 0 | 1 |
| $W_3$ | 0 | 1 | 0 | 1 |
| $W_4$ | 1 | 1 | 0 | 1 |

Figure 10—Associative memory operation

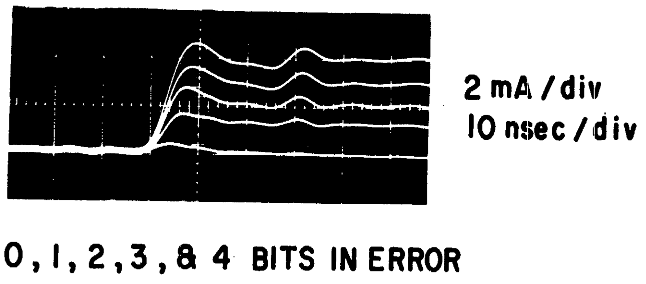


Figure 11—Analog mismatch signal

value of mismatch current. Associative memory operation is best illustrated by referring to Figure 10 which shows the contents of the memory as well as mismatch current waveforms generated for three different search criteria. The result of the first search for contents 0001 correctly indicate a match in word two only. Note that the mismatch current in word four, which has two bits in error, is in excess of 2 mA while that in words one and three is only 1 mA corresponding to a single incorrect bit. The second and third photographs again show correctly the proper mismatch waveforms for search criterion ϕ001 and ϕϕ01, the last of which correctly shows a match for all four words if the first two bits are ignored.

The additional feature of "proximity" matching alluded to previously is shown more clearly in Figure 11 where the mismatch output is shown for zero, one two, three and four bits in error in a given word. Use of analog detection circuitry at this point will greatly enhance the utility of resultant associative memory system.

Although the work discussed here is of a research and development nature and the volumes of arrays obtained are relatively small, it would be remiss at this point to avoid any discussion of yield, an all-important factor in integrated electronics. It is perhaps even more difficult to discuss this area when one considers the fact that in this new technology, a number of problem areas had to be overcome before *any* complex arrays were obtained. From that point on, however, the results were extremely encouraging as yields of 30 to 50 percent on packaged arrays were obtained ith extremely reproducible device characteristics. These represent relatively high yields when compared with monolithic MOS circuits of comparable complexity. It is believed, again with limited data, that these figures are a direct result of SOS technology wherein the amount of critical silicon is limited to the channel regions of the devices, and that yield depends only on this area rather than on total chip area as in monolithic circuits. In the 16-bit associative array, the critical area described represents 180 square mils whereas the total chip size is in excess of 5000 square mils, so that significant improvements in yield should and do result.

## SUMMARY

System, circuit, and device processing concepts have been developed and have resulted in the successful realization of high performance silicon-on-sapphire associative memory arrays. Features of the array include high speed current sensing for mismatch detection and non-destructive read out. The complementary MOS process sequence utilized in the array fabrication resulted in yields as high as 50 percent and produced high quality complementary devices with field effect mobilities of 300 and 150 cm$^2$/volt-sec for electrons and holes, respectively. The drastic reduction of parasitic capacitance inherent in SOS technology combined with these device characteristics provides a performance level equivalent to the highest speed bipolar circuits while retaining all of the other desirable circuit and processing

features of MOS arrays.

## ACKNOWLEDGMENTS

The authors gratefully acknowledge the contributions of D. J. Dumin and his associates in providing the silicon-on-sapphire films used in this work. A. O'Toole, J. Sokoloski and Mrs. R. Gilchrist are responsible for the array processing while W. Salt and Mrs. B. Denton diced, mounted, and bonded the sapphire pellets. Thanks are due to G. Cullen and G. Gottlieb for supplying the material from which the low temperature processing sequence evolved. R. Powlus contributed to the initial phase of associative array design. J. Previte of Rome Air Development Center deserves considerable credit for recognizing the potential benefits of thin film active devices.

## REFERENCES

1 J ALLISON J BURNS F HEIMAN
*Silicon-on-sapphire complementary MOS memory cells*
IEEE J Solid State Circuits Dec 1967

2 C Y WRIGLEY L J KROKO
*Properties of the silicon-sapphire interface in heteroepitaxy*
Electrochemical Society Semiconductor Silicon Abstracts May 1969

3 D DUMIN
*Deformation of and stress in epitaxial silicon films on single crystal sapphire*
J Appl Phys Vol 36 1965 2700

4 E C ROSS G WARFIELD
*Effects of oxidation on electrical characteristics of silicon-on-sapphire*
J Appl Phys Vol 40 1969 2339

5 F HEIMAN
*Thin film silicon-on-sapphire deep depletion MOS transistors*
IEEE Trans on Electron Devices Vol 13 1966 855

# A main frame semiconductor memory for fourth generation computers

*by* THOMAS W. HART, JR., DURRELL W. HILLIS,
JOHN MARLEY, ROBERT C. LUTZ and CHARLES R. HOFFMAN

*MOTOROLA, SPD*
Phoenix, Arizona

## INTRODUCTION

It has been obvious for several years that Large Scale Integration could be applied to memories. Memories offer several advantages in that a large volume of one type of device can be manufactured, and that the design can be optimized for one application. There exists a wide spectrum of memory product areas with varying size, costs, speed and enviromental performance. Most of these application areas are presently serviced by various forms of magnetic storage.

Semiconductor memories have been encroaching into some of these areas. First, the "scratchpad" was replaced by semiconductor memories yielding a better performance at lower cost. Secondly, the small buffer memories are now being implemented by various forms of semiconductor storage, mainly by MOS shift registers. Large very high speed semiconductor buffers are being built for large systems such as the IBM 360/85 to effect a hardware performance increase of slower core main memories.

It is felt that the advent of an all semiconductor main frame memory is fast approaching. The initial market penetration will be in the high performance area (100-300 ns) replacing flat-film memory designs where costs per bit are quite high. Eventually, most memory application areas will be vulnerable to semiconductor implementation on a price and performance basis. This paper will describe a memory module which will be used as a building block to implement high performance memories in the next generation of computers.

### *Module description*

Under various engineering and marketing constraints, a module building block concept evolved. This module in its general form contains 8192 bits. Interface to and from the module is performed with standard current-mode logic levels. MECL levels were chosen because that logic family provides the fastest interface when connecting many modules into a large memory system. Also, most of the customers and potential customers working on high speed systems are using some form of current-mode logic. In any event, it is not difficult to interface from other logic families to MECL levels.

By varying the logical connections to the module, an organization of 8192×1, 4096×2, 2048×4, or 1024×8, can be obtained. Figure 1 shows a block diagram of the module. Addressing is binary. Inputs and outputs may be bussed with other modules for expansion of the number of locations in a memory system. No complicated timing is necessary to operate the module. When an address is applied, the contents of the specified address will appear at the output terminals within 85 ns and remain until a new address is presented. Writing in a specified location is accomplished by pulsing the write enable line after the address and data have been presented. The module can be cycled every 100 ns.

The memory module uses p-channel MOS flip-flops for storage. Address decoding, word drive, sense, and digit drive are accomplished with bipolar circuits. This combination results in a low power, low cost

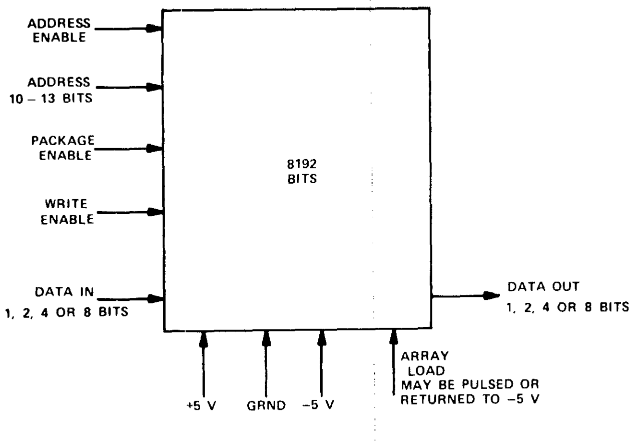

ADDRESS ENABLE
ADDRESS 10 - 13 BITS
PACKAGE ENABLE
WRITE ENABLE
DATA IN 1, 2, 4 OR 8 BITS
8192 BITS
DATA OUT 1, 2, 4 OR 8 BITS
+5 V
GRND
-5 V
ARRAY LOAD MAY BE PULSED OR RETURNED TO -5 V

Figure 1—Block diagram 8192 bit module

memory array, while retaining high speed module performance because of the bipolar circuits. The memory array itself contributes only a small fraction of the time used in a memory cycle (see timing diagram, Figure 14). The cycle-time is mainly determined by the bipolar circuits peripheral to the MOS—storage array.

The memory module was designed to operate on ±5v power supplies since these are fairly standard in integrated logic circuits. Total power dissipation is about six watts. While readily accomplished, no attempt was made to reduce power by various switching and pulse powering schemes since this level of power density can be easily handled in most applications by forced air cooling.

*Electrical description*

The module is a multipackage hybrid assembly. Four different integrated circuits are used to construct the module. These chips are (1) 256 bit MOS storage array, (2) Array Select Circuit, (3) Word Decode and Drive Circuit, and (4) Sense-Digit Circuit. The complete module has 32 Storage Arrays, four Array Select Circuits, two Word Drive Circuits, and four Sense-Digit Circuits.

*Storage array chip*

A block diagram of the 256 bit MOS Storage Array chip is shown in Figure 2. The array is organized in 2D fashion as 32 words × 8 bits. The linear select organization minimizes the number of devices per storage cell and also the number of inter-connections on the chip. Unfortunately, linear select organization causes some complications in packaging. These problems are circumvented here by placing sense line switches on the same chip as the array. This provides two benefits. First, additional addressing can be performed with the sense switches improving decoder

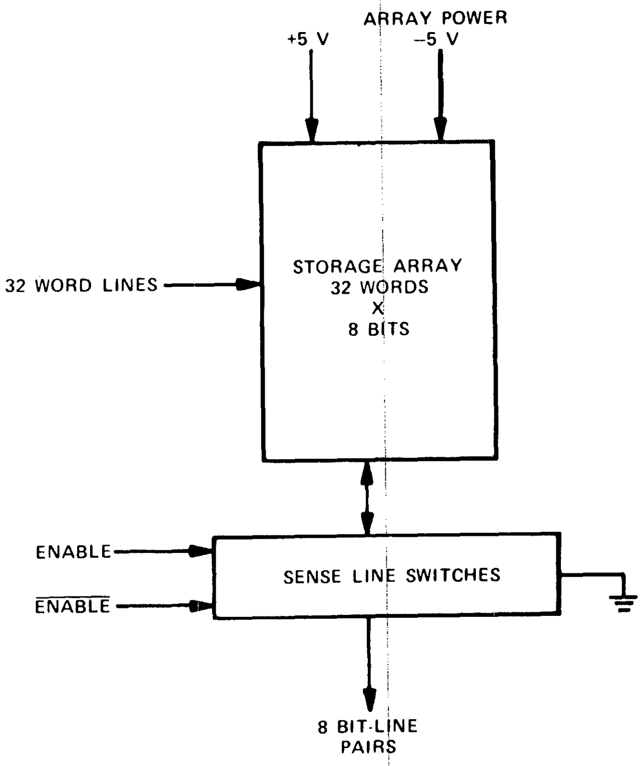


Figure 2—Block diagram MOS storage array

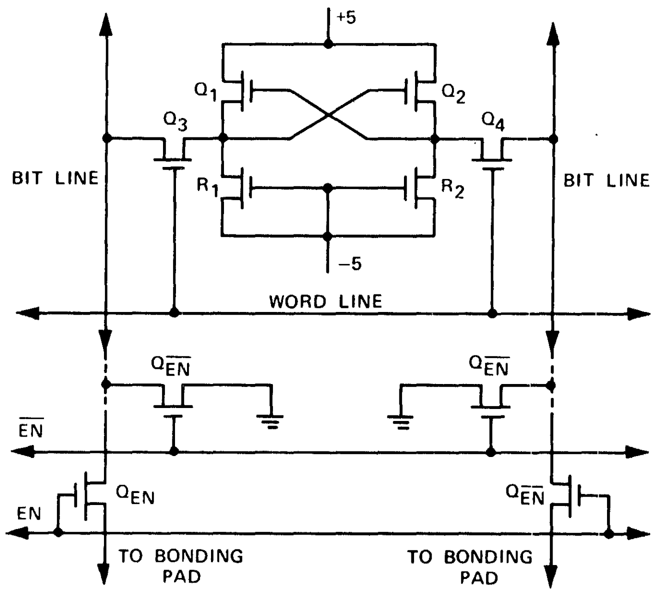


Figure 3—Storage cell circuit schematic

efficiency. Second, the internal capacitance of the bit-lines can be isolated from the external bit lines by the sense switches, substantially improving the sense loop time constant.

A schematic of a storage cell and the MOS sense-switches at the end of the bit line is shown in Figure 3. $Q_1$ and $Q_2$ are the active devices of the flip-flop, R1 and R2 are the flip-flop load devices, and $Q_3$ and $Q_4$ are the series gating devices which connect a selected cell to the bit line pair. Each bit line has a transistor $Q_{EN}$ in series with the connection to the bonding pad and a transistor $\overline{Q_{EN}}$ which terminates the bit line to ground when $\overline{Q_{EN}}$ is on. The geometries of the active devices are designed to provide a sense current of 80 microamperes under worst case processing and operating conditions. The load resistor device geometries determine the standby power dissipation of the chip which in this case is about 40 milliwatts.

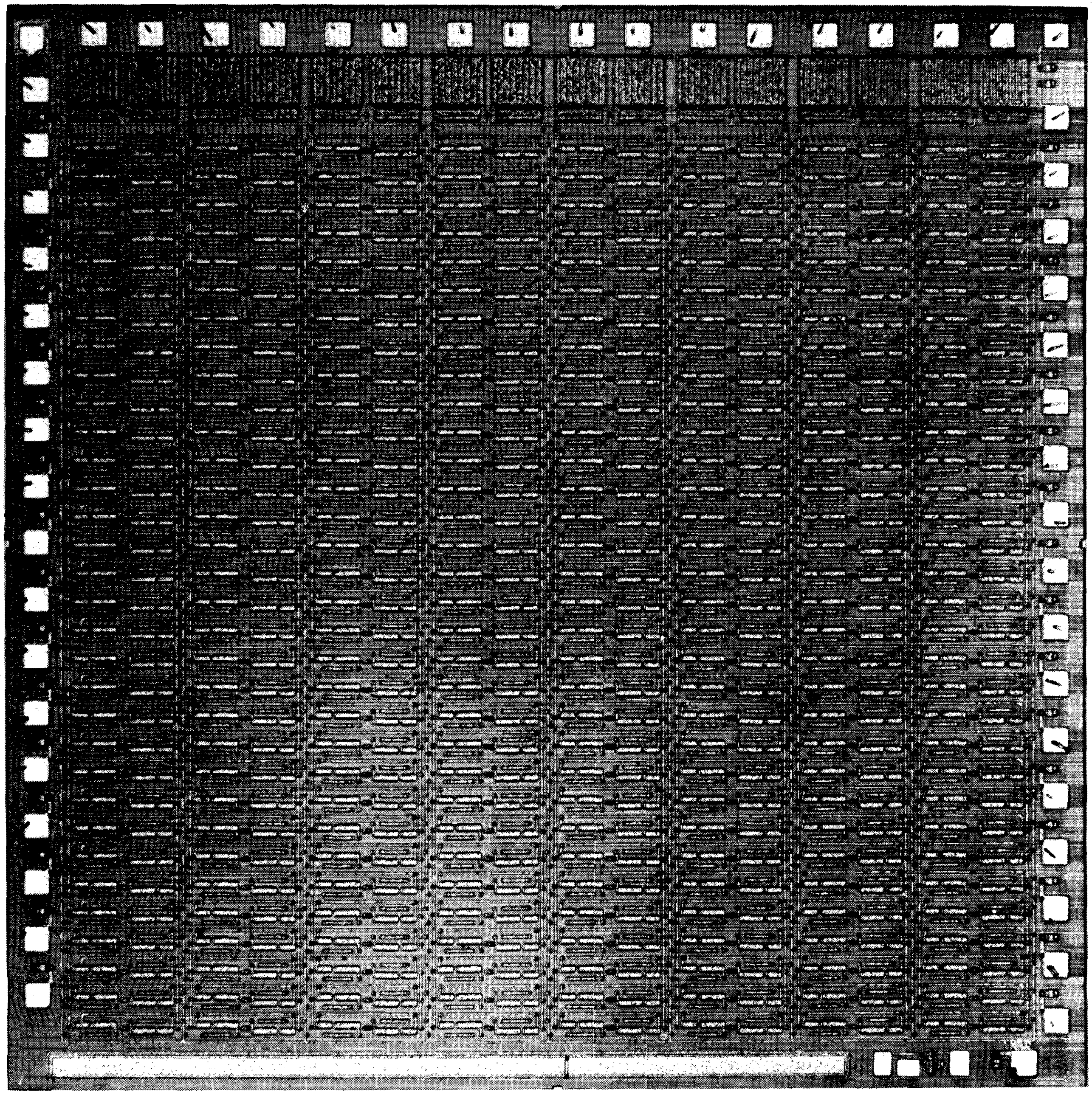


Figure 4—256 Bit MOS storage array

Figure 4 is a photomicrograph of the chip. The dimensions of the chip are 138 mils × 141 mils. A low threshold process using <100> material is used. The substrate serves as the buss for the +5 volt supply.

One layer of metal interconnection is used. A high concentration P-diffusion (15-20 Ω/square) is used for crossunders so as to minimize series resistance. In the layout the bit lines have no crossunders. The word lines have nine crossunders. The resistance of these crossunders and capacitance associated with the word gates on the memory cell form an RC delay line. In this design the delay is about 2.5 ns.

*Chip selection circuit*

A bipolar circuit which decodes three binary bits is used to select one of eight MOS Storage Array Chips. Each of the output driver stages provides the complimentary signals EN and $\overline{\text{EN}}$ necessary to drive the sense switches on the MOS Storage Array. Additional inputs to the chip-selection Circuit are provided to select groups of eight arrays.

Emitter Coupled Logic (ECL) input signals are translated to saturated logic which is referenced to the negative supply (−5.0). The complimentary output stages provide logic levels near the positive (+5) and negative (−5) supplies for driving the MOS sense switches. Block and Logic Diagrams are shown in Figures 5 and 6.

*Memory package*

Eight MOS Storage Array chips and one chip selection circuit are contained in a 1.25 inch square memory package. Interconnection of these nine chips is made by a beam lead laminate as described later in this paper. Each memory package contains a total of 2048 bits as shown in Figure 7. Four such packages form the storage portion of the 8192 bit memory module. This assembly of four packages results in a total capacitance buildup of 250 picofarads on the word lines and 70 picofarads on the sense-digit lines.

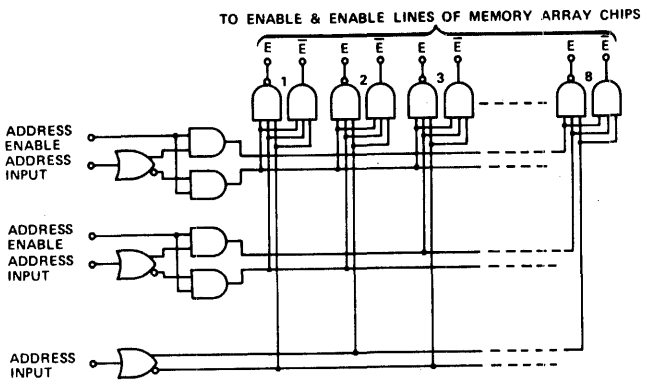


Figure 6—Chip select circuit logic

*Decoding word driver*

Selection of the storage array word lines is accomplished by a bipolar circuit which decodes four address bits and drives one out of sixteen word lines. As in the Chip Selection Circuit, ECL input signals are translated to saturated logic whose outputs provide logic levels near +5 and −5 volts. Block and logic diagrams are shown in Figures 8 and 9. Two of these chips are packaged in a 1.25 inch square package similar to the memory array package except that interconnection within the package is made with a thick film metalization and wire bonds. Two address enable inputs are provided. One is used as a master enable and the other is used as a one bit decode to select one or the

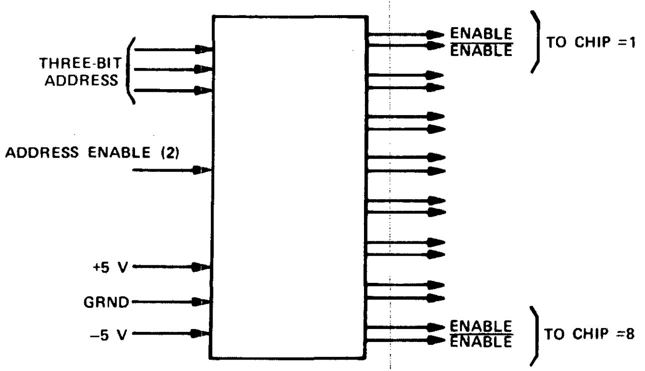


Figure 5—Chip select circuit

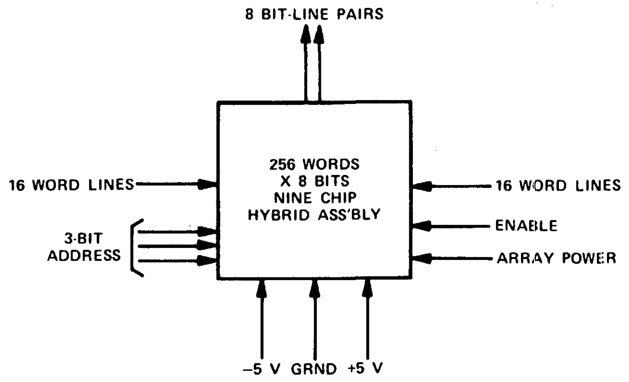


Figure 7—Memory package

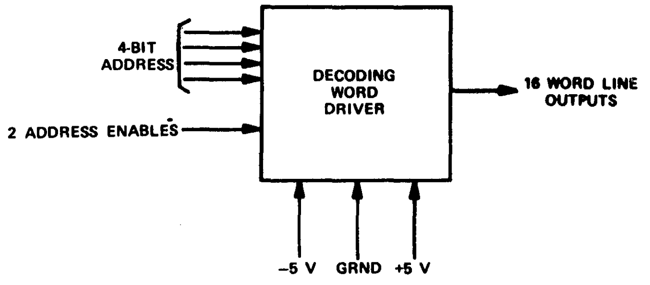


Figure 8—Decoding word driver chip

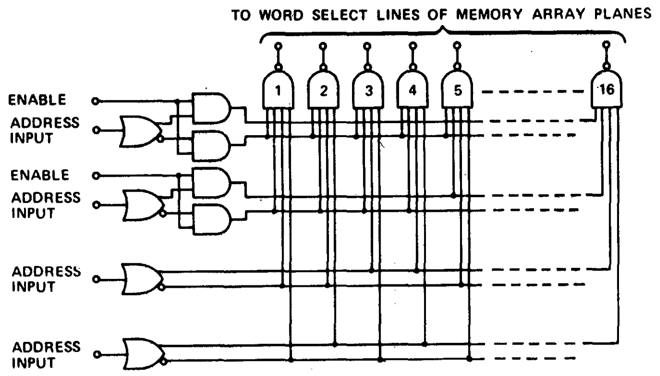


Figure 9—Decoding word driver chip logic

other of two Decoding Word Driver chips sharing the same package. A block diagram of this package is shown in Figure 10.

*Sense amplifier-digit driver*

The sense amplifier-digit driver subassembly contains four identical sense amplifier-digit driver integrated circuit chips. Each chip receives and sends read and write signals to the MOS storage array, accepts ECL level data input and data enable signals, and generates ECL data output signals.

The purpose of each chip is, of course, twofold. First, when it has been properly enabled for writing, it must transmit a write signal to appropriate bit(s) of the selected word in the storage array. Second, when properly enabled, it must sense the storage cell currents of the selected word and translate them to ECL signals at the data output terminals.

The logic diagram shown in Figure 11 is functionally equivalent to the sense amplifier-digit driver circuit. In addition to showing the basic sense amplifier, digit driver, and gate blocks of the sense amplifier-digit driver chip, Figure 11 also shows the existence of a bit line recovery circuit. The purpose of this circuit is to rapidly return all bit line voltages to zero, immediately after each write operation.

To thoroughly understand the sense amplifier-digit driver logical organization, consider the sequence of events which must occur to perform the read and write operations.

To accomplish a write operation, the desired input data is placed at the DATA IN terminals of the chip. The data is enabled by a coincidence of logical zeroes at the DATA ENABLE inputs. When the WRITE ENABLE input is forced to a logical zero, one of the bit line voltage drivers in each half of the circuit drives one line of each bit line pair to approximately +4v. This voltage impressed on a bit line accomplishes the write in the storage array. The leading negative edge of the WRITE ENABLE signal also sets the recovery circuit flip-flop. The following positive edge of the WRITE ENABLE signal turns the digit driver(s) off and turns the recovery circuit driver on. When recovery of all bit lines is accomplished, the recovery circuit flip-flop resets and the recovery circuit driver is shut off. Both the digit driver and the recovery circuit driver are designed to exhibit a very high output

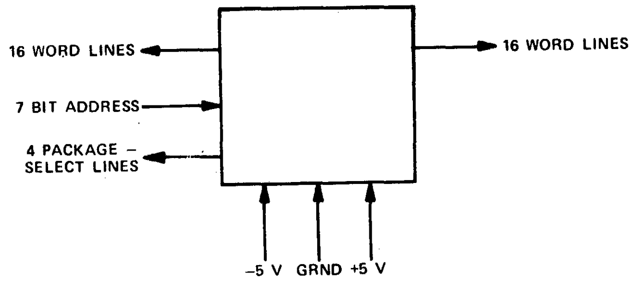
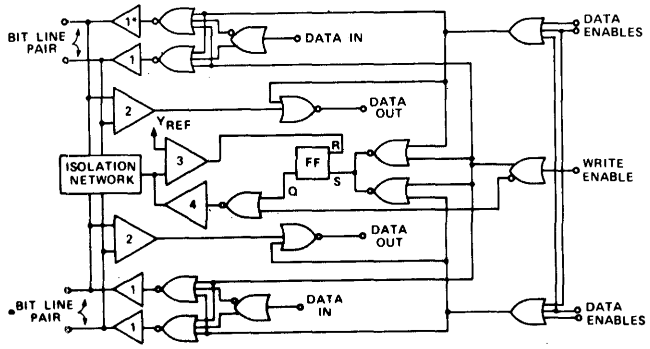


Figure 10—Decoding word driver package



NOTES THE NUMBERED BLOCKS CORRESPOND TO THE FOLLOWING: 1 BIT LINE VOLTAGE DRIVER, 2 SENSE AMPLIFIER, 3 VOLTAGE COMPARATOR, 4 BIT LINE RECOVERY CURRENT DRIVER

Figure 11—Sense amplifier/digit driver chip logic

impedance when off, such that they do not interfere with the read operation.

Reading is accomplished by enabling either one or both halves of the chip with the DATA ENABLE signals. If the WRITE ENABLE is held at logical one, the bit line currents flow into the sense amplifier inputs. The sensed information is made available at the DATA OUT terminals. Since the I/O signals are ECL, uncommitted emitter outputs are used so that wired OR'ing of the positive going output signals is possible.

Figure 12 shows a block diagram of the sense amplifier-digit driver package. Since the DATA OUT signals from all four sense amplifier-digit driver chips can be OR'ed, various connections of the DATA ENABLE and DATA IN signals are possible. If the DATA ENABLES are connected for maximum decoding, a one-out-of-eight selection can be accomplished. With all eight DATA IN inputs and DATA OUT outputs strapped together, the module organization becomes 8192 words of one bit. Similarly, if all DATA ENABLES are tied together, each DATA IN and DATA OUT is used as a separate information channel, and the resultant module organization is 1024 words of eight bits. Other connections result in 4096 words of two bits, and 2048 words of four bits. These various connections occur external to the module. Hence, the sense amplifier-digit driver plane organization is the same regardless of the final module organization desired.

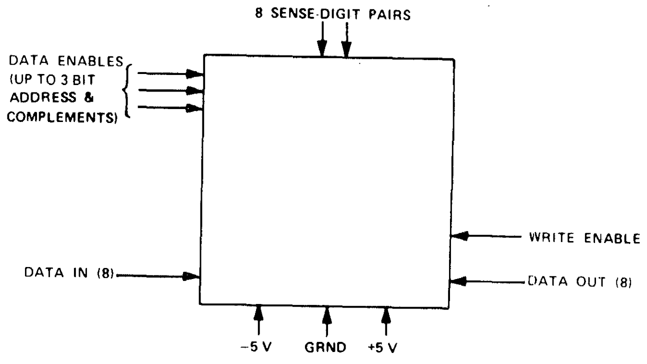


Figure 12—Sense-digit package

*Module electrical organization*

Figure 13 shows an integrated electrical schematic

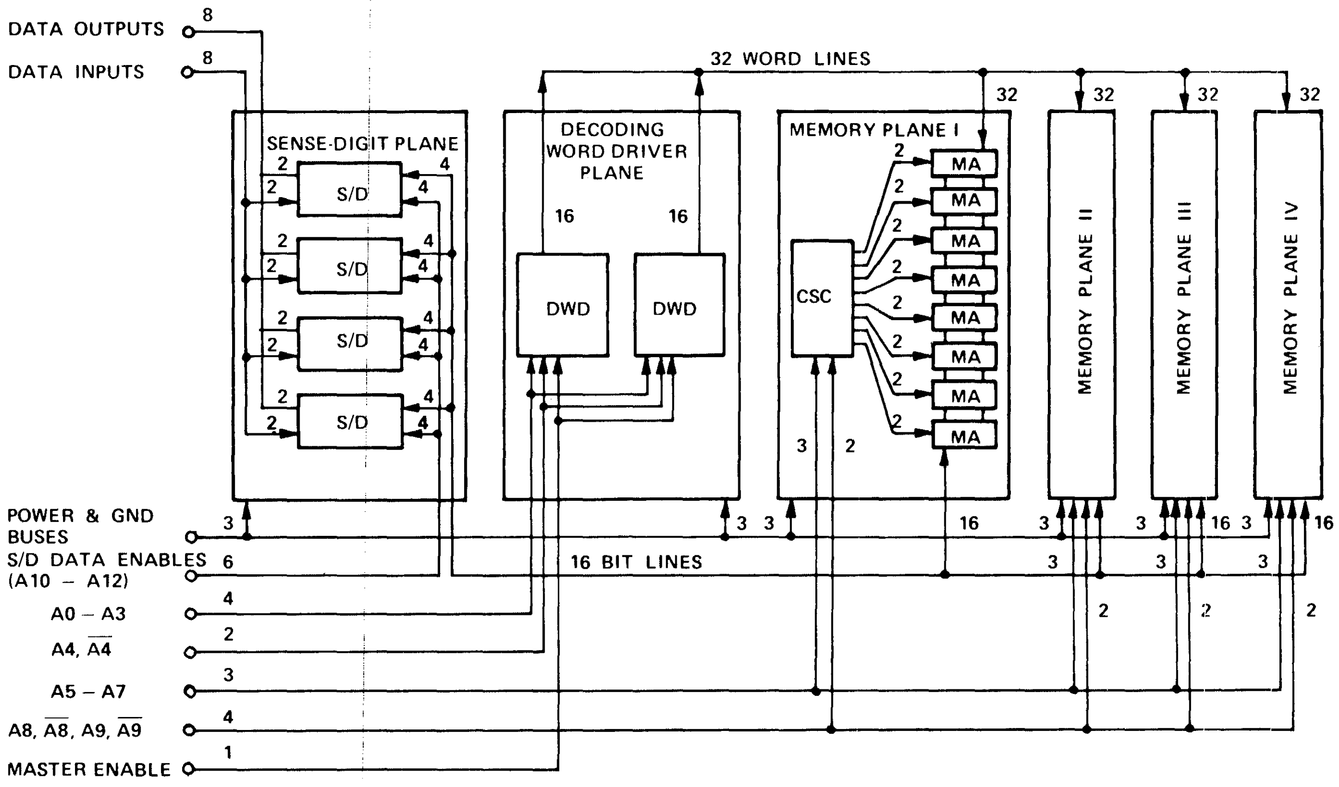


Figure 13—8192 bit memory stack-electrical organization

of the 8192 bit memory module. Notice that the package-to-package connections are accomplished by means of long parallel busses or rails which are formed when the six packages are stacked. Although these rails must be broken in some places to define the interconnection, no jumper wires, etc., need be added.

There are three distinct methods of addressing shown in Figure 13:

1. Address bits A0, A1, A2, and A3 are bussed in parallel to the two decoding word driver chips (DWD). Since each DWD is a one out of sixteen decoder, an additional address bit (A4-$\overline{\text{A4}}$) is decoded on the enable inputs of the DWD's. Hence, the decoding word driver package functions as a one out of thirty-two decoder, selecting only one of 32 word select lines to enable in the memory array. Note that all word select lines in the thirty-two MOS array chips are wired in parallel.

Address bits A5, A6, A7 drive the chip select circuits (CSC) in the memory array packages. The CSC drives the ENABLE and $\overline{\text{ENABLE}}$ of memory array's sense line switch, and select one of the eight array chips with each memory array package. To complete the selection of an eight bit word on one MOS array one of the four memory planes must be uniquely enabled. The two enable inputs of each chip select circuit, (CSC) are connected to address bit A8 and A9 or their complements to perform the plane selection. Connection to the correct two signals is accomplished by providing A8, $\overline{\text{A8}}$, A9, and $\overline{\text{A9}}$ at the edge of all memory array packages and breaking the bus connection with two of the signals.

These ten bits (A0-A9) select one eight bit word out of the 1024 word array. Subdivision of the eight bit word into four, two, or one bit words is accomplished by strapping lines (DE1-DE6) together in specific combinations. The sense amplifier package can accept zero, one, two, or three address bits depending on the module organization desired.

Once a single eight bit word in the storage array has been addressed, sixteen low impedance paths (eight pairs of bit lines) exist between the four sense amplifier-digit driver chips and the eight MOS array cells. Reading is accomplished by sensing the storage cell currents while holding the bit lines at approximately ground; writing by forcing selected bit lines to a positive voltage.

In addition to the standard I/O and address channels, a DWD ENABLE and/or CSC ENABLE are brought out to allow for further addressing or for eliminating skew in the address signals.

### *Module timing*

Figure 14 is a timing diagram of the 8192 bit module. The diagram is organized into three sections: (1) the basic addressing and enabling common to both read and write cycles. (2) the basic read cycle, and (3) the basic write cycle. The diagram illustrates how the various propagation and charging delays add to form the minimum cycle times.

Since all word select and bit lines are parallel connected to all thirty-two array chips, the capacitances associated with these lines are high (approximately 250 pf on word lines, 70 pf on bit lines). Therefore, charging times become a significant portion of the memory cycle time.

The labels on the diagram are generally self explanatory. The comments below explain some of the special features shown.

Notice that if desired, the sense amplifier-digit driver data enable inputs (DE1-DE6) can arrive at the module terminals some 15 ns later than the address signal (A0-A9). Therefore, additional levels of decoding logic may be added in series with these inputs without slowing the cycle time.

During the read cycle the sensed information must be strobed out toward the end of the cycle, as the information on the sense amplifier-digit driver DATA OUTPUTS prior to that time is the stored information in the previously addressed location or undefined. Bit line recovery after writing is overlapped into the next cycle.

### *Packaging*

The 8192 Bit Memory Module is an assembly of four 2048 Bit Memory Array packages, a word driver package, and a sense digit package. Each of these packages are 1.25 inches square with 17 leads on 50 mil centers on each side. The electrical organization is such that the packages can be stacked one above the other with the leads bussed. After assembly and test of the individual packages are complete, the individual leads are cut and formed. The packages are then placed in a mechanical holder, and wave soldered one side at a time. A molded header is then mechanically and electrically attached to the base of the stacked assembly of packages. The header includes guide slots so that the ends of the ribbon leads can be inserted into a simple etched wiring board nested within the header. The 42 signal pins of the header are arranged in a 1.6 inch square on 100 mil centers. This choice of pin form factor permits established printed circuit board technology to be employed by the user.

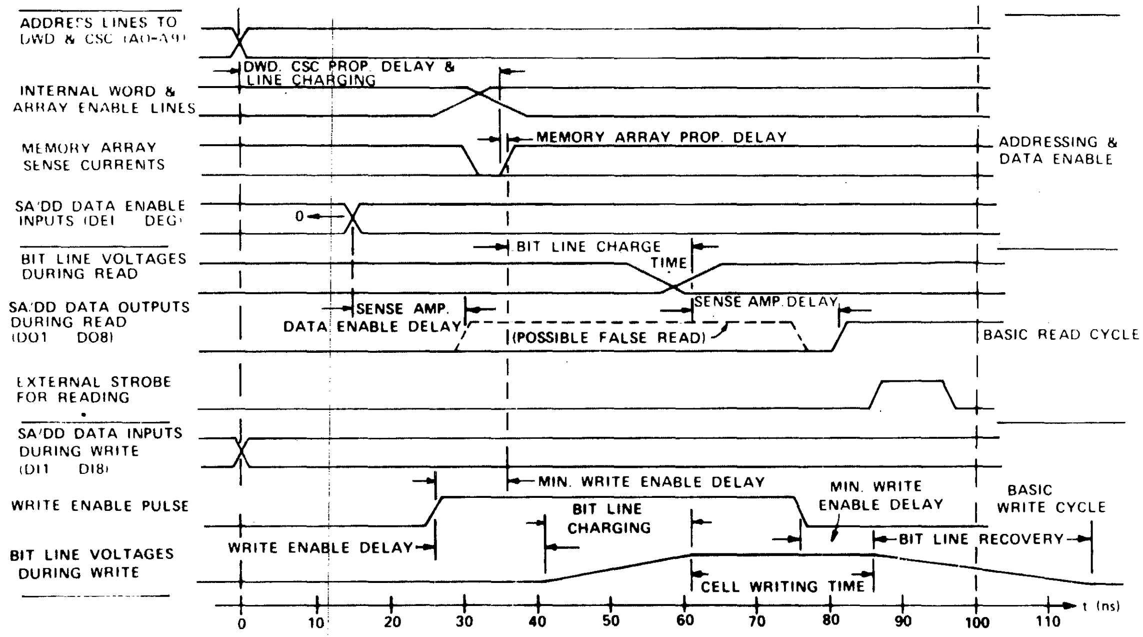

ADDRESS LINES TO DWD & CSC (A0-A9)
INTERNAL WORD & ARRAY ENABLE LINES
MEMORY ARRAY SENSE CURRENTS
SA/DD DATA ENABLE INPUTS (DEI DEO)
BIT LINE VOLTAGES DURING READ
SA/DD DATA OUTPUTS DURING READ (DO1 DO8)
EXTERNAL STROBE FOR READING
SA/DD DATA INPUTS DURING WRITE (DI1 DI8)
WRITE ENABLE PULSE
BIT LINE VOLTAGES DURING WRITE
DWD. CSC PROP. DELAY & LINE CHARGING
MEMORY ARRAY PROP. DELAY
ADDRESSING & DATA ENABLE
0
BIT LINE CHARGE TIME
SENSE AMP. DATA ENABLE DELAY
(POSSIBLE FALSE READ)
SENSE AMP. DELAY
BASIC READ CYCLE
MIN. WRITE ENABLE DELAY
MIN. WRITE ENABLE DELAY
BASIC WRITE CYCLE
BIT LINE CHARGING
WRITE ENABLE DELAY
BIT LINE RECOVERY
CELL WRITING TIME
0
10
20
30
40
50
60
70
80
90
100
110
t (ns)

Figure 14—Timing diagram - 8192 bit MOS stack memory

The heat generated by this module is approximately 6 watts. The volume of the module assembly including the 42 pin header and plug is 1.75 × 1.75 × 2.0 inches or six cubic inches. Operational bit density is thus over 1300 bits per cubic inch. It is felt that these two counteracting factors are fairly well balanced to each other by this module design concept.

### *Package interconnect*

The Sense-Digit and Word Driver packages are simply one layer thick film metal patterned ceramic packages with connections being made with flying wire leads. The memory array package is much more densely populated and uses a new technology of interconnect.

Interconnect technologies currently available are multilayer ceramic, multi-layer surface deposition, or multi-layer "add on" laminate. Ceramic multi-layer was not selected for three reasons. The high dielectric constant of alumina raises the distributed capacitance to levels which threaten system speed requirements. The many vias required for intra-layer connection cannot be placed on close enough centers to be compatible with the desired cell densities. Finally, the length of buried conductors used for power distribution exhibit higher resistance than is desired for low noise level operation.

Use of multi-layer surface depositions on a suitable package substrate has been avoided because of yield problems of dielectric defects in the presence of many crossovers, and the presence of deposition interfaces at each via buildup location. Surface deposition of a single low-impedance thick film pattern on the package substrate has been utilized for power distribution to the IC chips within the container.

### *Interconnect laminate*

The interconnection is implemented by a separate part called an interconnect laminate. The dielectric core of this laminate is 1 mil polyimide film and exhibits the following characteristics:

1. Physically and electrically stable dielectric through the range of $-65°C$ to $+450°C$.
2. Dielectric constant of 3.5.
3. Pinhole free and a voltage breakdown rating of 7000 volts at one mil thickness.

Through a series of precisely-registered artwork and photo-chemical cycles, the two-layer X-Y interconnections are formed to the following standards:

1. Via size of 1.5 to 2.0 mils diameter on seven mil centers.
2. Via lands or caps are 4 × 5 mils.
3. Conductor widths are three mils and conductors are spaced on five mil centers.

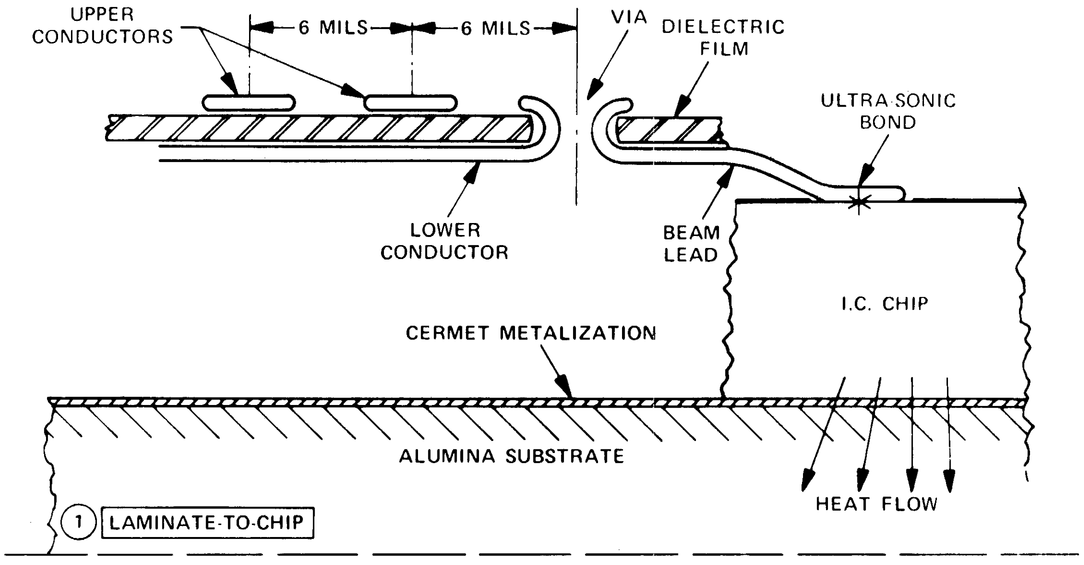


Figure 15—Interconnect cross-section

4. Conductors on both sides and the vias are electroplated as a single structure having no metallurgical interfaces at the vias.

These standards of fabrication provide an interconnect system which is compact and comparable in geometry details to the bond pads and spacing used by MOTOROLA on its IC chip products. Capacitance measurements of typical center conductors to the sum of grounded neighboring conductors, using the above dimensional and material standards, read about 2.15 picofarads per inch of length while dc resistance of typical conductors measures 0.40 ohms per inch of length.

*Bonding*

Of the three primary methods for connecting signal lines to the IC chips pads (wires, bumps, and beam leads), a system employing beam leads is used. These beams are integrated into the laminate plating rather than using the more customary method of integrating the beams into the IC chip. This was done to avoid additional processing steps to the already complex wafer. Not only is the silicon wafer yield protected, but a packaging thermal advantage is obtained by being able to beamlead bond "face up" against the IC chips. The rear surface of the chips is then mechanically secured to the substrate base, assuring low thermal resistance.

A cross-section diagram showing the features of the inter-connect laminate, the ultrasonic "face up" bonding technique, and the heat sinking capability to the power-carrying cermet metalized alumina substrate are illustrated in Figure 15. The face-up technique permits bonding to the chip one beam lead at a time

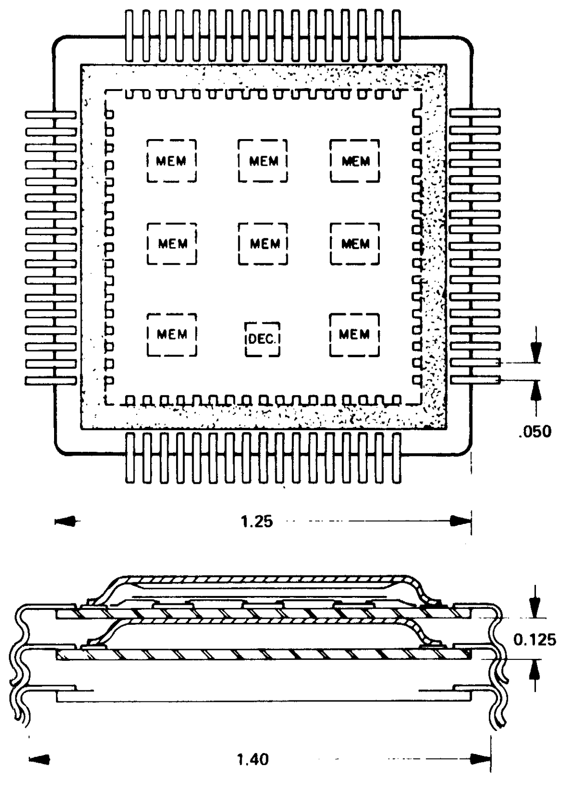


Figure 16—Packaging for memory stack

or one chip at a time. It also permits quite stringent quality control measures to be implemented since the beams can be examined individually.

### *Memory package*

A sketch of the package which is being used is illustrated in Figure 16. It consists of a 1.28 inch square, 96 percent alumina base, which is metallized to a custom pattern containing 68 metal film leads which go under a glass-sealed side wall. The base of the usable 1.0 square inch interior contains the power distribution pattern. The headroom within the package is 60 mils.

As can be seen by inspection of the figure, the area occupied by the IC memory chips and the control chip is approximately 25 percent of the area, the remaining area being used for the X-Y interconnect and exit bond functions.

In the assembly cycle, a total of 448 beams leads are bonded to the IC chips which is half of the bonds required by wire bonding techniques. The laminate contains 480 electrically active plated feed throughs. Larger beam leads are employed to connect the interconnection laminate to the exit bond pads and the power distribution. A total of 73 such bonds are required. In the computer program which generated the interconnect laminate artwork master sets, approximately 1400 conductor track segments instructions were generated. The cover is alloyed to the package subassembly after precap testing. The result is a memory component containing 2048 MOS memory cells and having only 68 leads to the outside world.

## CONCLUSION

A high performance memory module has been described which is suitable for use as a building block for large mainframe memories. Mass production of this memory module is planned. Costs per bit of a memory system using these modules as basic building blocks will be much lower than that of other technologies giving a similar performance. In the near future the competitive pressure of semiconductor memories will be felt in most performance ranges. Magnetics watch out!

# A new approach to memory and logic-cylindrical domain devices

*by* A. H. BOBECK, R. F. FISCHER and A. J. PERNESKI

*Bell Telephone Laboratories*
Murray Hill, New Jersey

## INTRODUCTION

Magnetic domain behavior in single crystal magnetic oxides has been studied extensively over the last several decades. These investigations, both theoretical and experimental, are an attempt to better understand these materials and their complex domain structures. Recently single crystal oxides have been utilized in memory and logic devices. This paper will update work on cylindrical domains in orthoferrites first published in 1967 and later discussed at the 1968 and 1969 Intermag Conferences.[1,2,3]

A cylindrical domain, sometimes referred to as a bubble, is a localized high energy magnetic state. Such a domain is stable and resists any attempt to deform it. Domains can be moved about in much the same way as a charged particle. A domain can be moved one domain diameter in less than 100 nanoseconds thus indicating that data rates in excess of $10^6$ bits/sec can be realized in this technology. As yet no upper limit to the cylindrical domain velocity has been found experimentally.

Sucessful device utilization of cylindrical domains depends upon developing techniques for generating propagating, interacting and detecting these domains. Domains can be generated by sectioning an existing domain into halves. Each new domain can be considered as an information input if the splitting operation is selectively controlled. A stream of domains, fed into a propagation channel and transmitted to an output point, can be detected by optical, Hall or induced voltage readout. Although all these readout techniques have been studied only induced voltage readout will be detailed in this paper.

A new class of materials, the orthoferrites,[4,5] are now available which, in addition to supporting cylindrical domains at densities approaching $10^6$ per square inch, have the combined properties of high nucleation fields (so domains will not spontaneously appear), low domain wall coercivities, and high domain wall mobilities. A description of the general properties of cylindrical domains[6,7] in orthoferrites is followed by a section on the behavior of domains in gradient fields. Conductor circuits, "angelfish" circuits[8] and in-plane rotating field circuits[9] are presented as general methods to propagate domains. Finally the relevance of domain wall devices to the computing field is discussed.

### *General observations*

If we take a thin platelet of orthoferrite above its Néel temperature and cool it to room temperature spontaneously nucleated serpentine-like strip domains will be present. Such a domain pattern, as seen in Figure 1, will usually include a number of single wall domains. A single wall domain can be identified by noting whether the wall which bounds it closes upon itself. If a prescribed magnetic field, the bias field, is applied normal to the surface of the platelet the single wall domains become cylindrical. An array of such domains is shown in Figure 2. The 1.7 mil thick platelet of $Sm_{.55}\ Tb_{.45}\ FeO_3$ osoferrite is subjected to a 42 Oe bias field.

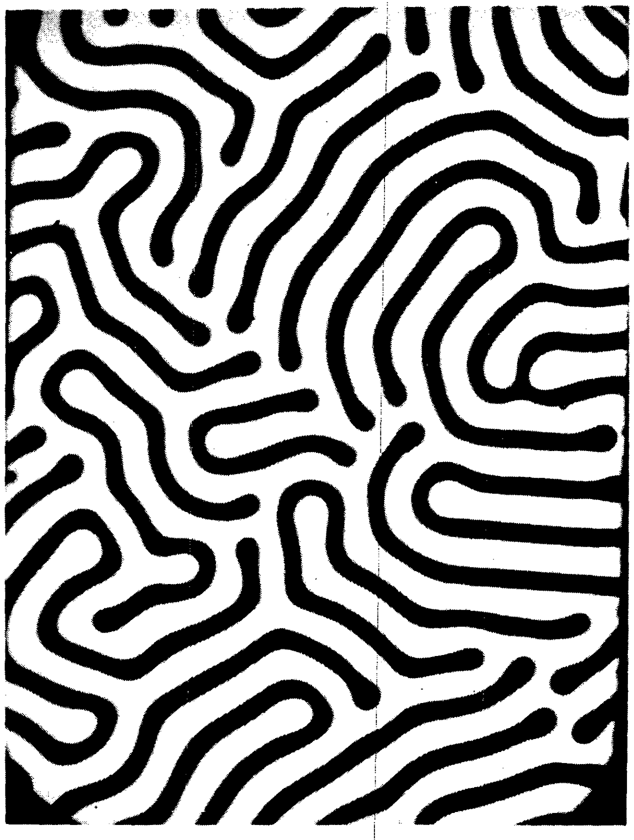


Figure 1—Strip domains, 1.5 mils in width, in a 1.7 mil thick platelet of $Sm_{.55}Tb_{.45}FeO_3$ orthoferrite viewed by Faraday effect. Note the single wall domains. Bias field is zero

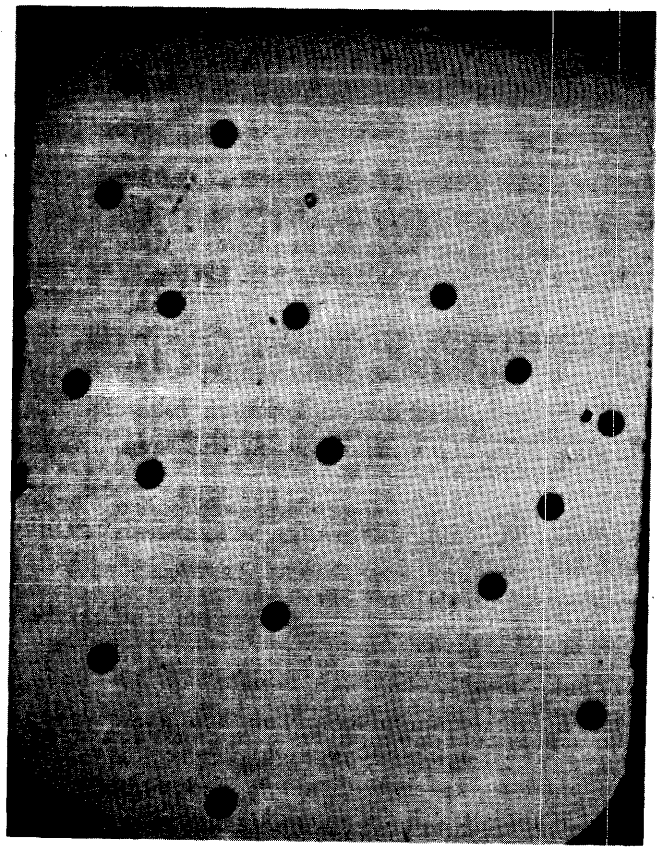


Figure 2—With a 42 Oe bias field the single wall strip domains of Figure 1 become cylinders each 1.8 mils in diameter

Those familiar with the earlier references recall that cylindrical domains are stable over a limited range of the bias field (typically 10 percent of $4\pi M_s$). An excess bias causes the domain to collapse inward. On the other hand as the bias is decreased the domains grow in size eventually reaching a diameter at which they become unstable to elliptical perturbations and then suddenly grow into long strip domains.

A strip domain can also be cut by energizing a conductor positioned in contact with the orthoferrite and intercepting the strip domain at right angles. For SmTb orthoferrite a current of 300 mA is sufficient. Later, in the discussions of conductor propagating circuits, a technique for splitting cylindrical domains will be presented.

*Manipulation of cylindrical domains—General*

Domains in orthoferrites are maintained in the preferred cylindrical form by an overall uniform bias field applied normal to the platelet surface. As discussed previously an increase in the bias field decreases the domain diameter and vice versa. Now consider the reaction of a cylindrical domain subjected to a nonuniform rather than a uniform field. The response will be complex and could involve a change in size, motion at a nonuniform rate or even the collapse of a domain. However, it is possible to treat the case in which a uniform gradient field is applied.

Consider, as shown in Figure 3, a cylindrical domain of diameter 2r in a uniform gradient field. The domain

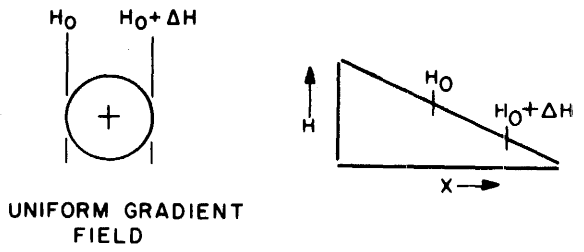


Figure 3—A cylindrical domain of diameter 2r positioned in a uniform gradient field

will experience a force attempting to move it toward a position of reduced bias. To overcome the wall coercivity, $H_c$, the following condition must be met:

$$\Delta H > 8H_c/\pi. \quad (1)$$

1

Furthermore, it can also be shown that the domain wal velocity, $v$, is given by

$$v(\text{cm/sec}) = \Delta H(\text{Oe})\ \mu(\text{cm/sec/Oe})/2 \quad (2)$$

where $\mu$ is the usual domain wall mobility.[6]

One method to see the effect of a gradient field is to interact one domain with another. In the case of domains widely separated the far field of a cylindrical domain can be approximated as that of a dipole and the following relationship derived (see Figure 4).

$$\frac{H_c}{4\pi M_s} = \frac{3\pi r_0^3 h}{8\ell_{12}^4}. \quad (3)$$

Equation (3) specified$\ell_{12}$, the minimum stable separation between a pair of domains as they repel one another because of their mutual gradient fields.

Finally it has been found useful to interact high permeability magnetic film patterns with cylindrical domains. Consider, for example, a matrix of permalloy dots positioned on the surface of an orthoferrite platelet. One finds, by experiment, that a cylindrical domain prefers a position in contact with the permalloy as shown in Figure 5. The permalloy dot diameters and separations have been chosen to be consistent with the stable cylindrical domain size in the orthoferrite under study. The dots serve as localized flux closure paths thereby reducing the magnetostatic energy. They provide a shift register, a memory array, etc., with well defined domain positions.

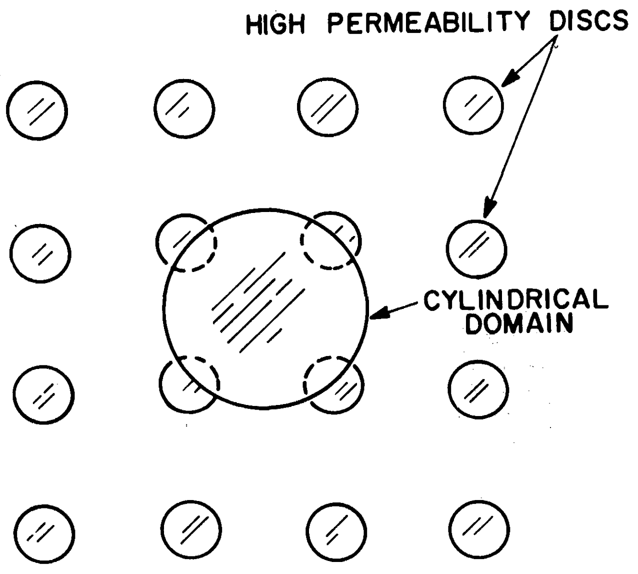


Figure 5—Interaction between a matrix of high permeability disks and a cylindrical domain

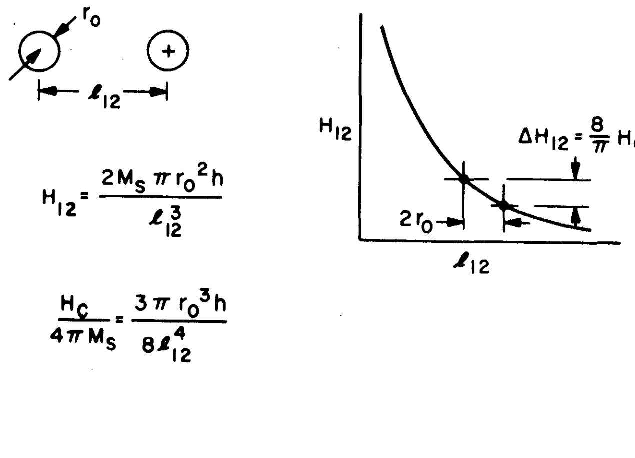

$r_0$
$\ell_{12}$
$H_{12} = \frac{2M_s \pi r_0^2 h}{\ell_{12}^3}$
$\frac{H_c}{4\pi M_s} = \frac{3\pi r_0^3 h}{8\ell_{12}^4}$
$H_{12}$
$\Delta H_{12} = \frac{8}{\pi} H_c$
$2r_0$
$\ell_{12}$

Figure 4—Two domains, mutually repelled in a material whose coercive force is $H_c$, reach a stable separation $\ell_{12}$.

*Conductor circuits*

In order to utilize cylindrical domains in shift registers, memories and logic circuits, we require motion in discrete steps at specific times. Therefore, highly localized fields are needed. Such fields can be produced by small conductive loops placed flat on a platelet surface. Since thin film techniques are used to fabricate the conductor circuits, a completely closed loop is not practical.

Figure 6 illustrates the basic conductive loop configuration and the resulting field profiles. These were obtained by measuring the fields produced by an expanded scale replica of the thin film circuits. The circuit dimensions were chosen to provide controlled motion of domains whose diameters range from 3.5 to 6 mils. In order for a domain to move to an adjacent loop it must initially be in contact with some portion of the positive gradient field produced by that loop.

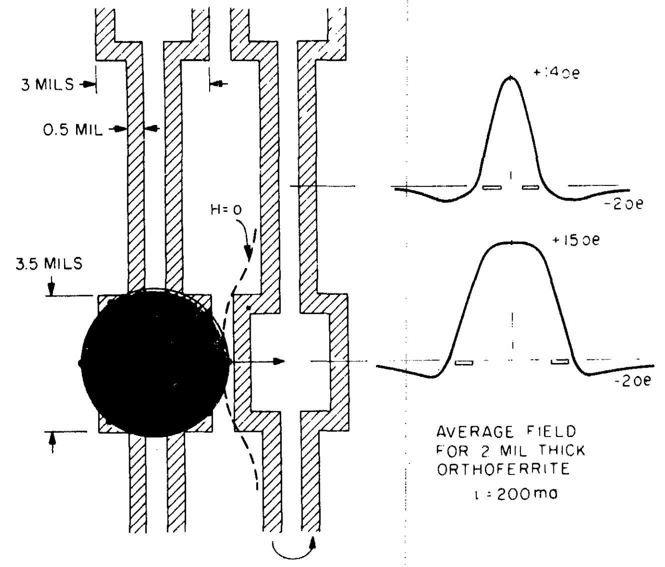


Figure 6—Conductor circuit used to propagate cylindrical domains and the resulting field profiles for 200 mA applied current

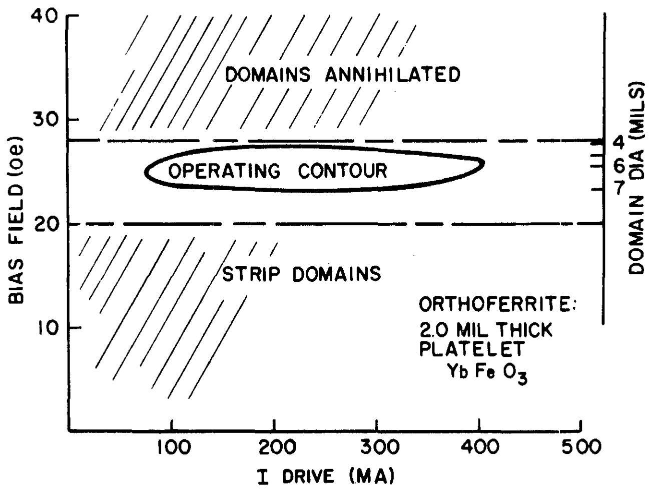


Figure 7—Quasi-static operating contour for 2-0 mil thick platelet of $YbFeO_3$.

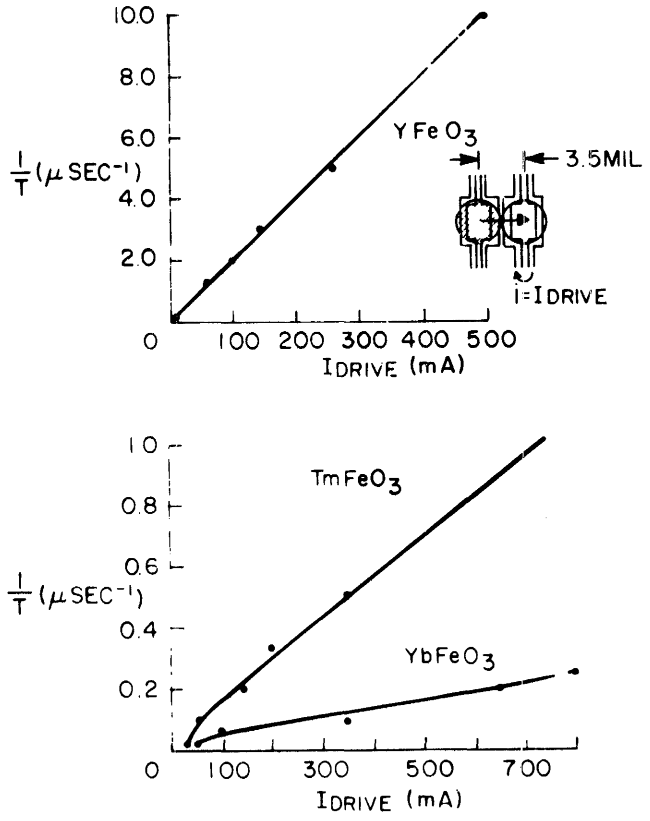


Figure 8—Functional velocity curves of $YFeO_3$, $TbFeO_3$ and $TmFeO_3$ platelets.

This puts a lower limit on the domain size. The limit of maximum domain size is reached when a disparity of domain to applied field area results in reduced control of the domain position.

The most important feature of the semiclosed conductive loop circuit is that the field is confined to an area consistent with that of a domain. Therefore, the field may far exceed the value which would transform a domain from a cylinder to a strip. The upper limit on this field, however, is that value which would stretch the cylindrical domain into the strip area defined by the connections between the loops.

The limits of the applied drive and bias fields are illustrated in Figure 7. The data was obtained using a 2.0 mil thick platelet of $YbFeO_3$ operated in a quasi-static fashion on a conductor pattern similar to that of Figure 10. The operating contour resides within the bias field extremes required to maintain a cylindrical domain. The position and size of the operating contour within the bias field boundaries is determined primarily by the range of domain sizes which the circuit can accommodate.

In Figure 8, velocity curves are given of domains in $YFeO_3$, $TmFeO_3$ and $YbFeO_3$ platelets. These are functional measurements obtained using the circuit of Figure 6. Rossol has shown that $YFeO_3$ exhibits an extraordinarily high mobility.[10] Functional velocity measurements of $YFeO_3$ have confirmed this. Data rates in excess of $3 \times 10^6$ bits/sec have been reached. A direct comparison of device speed and domain wall mobility cannot be made because of the complex nature of the field profile. Notice that threshold currents as

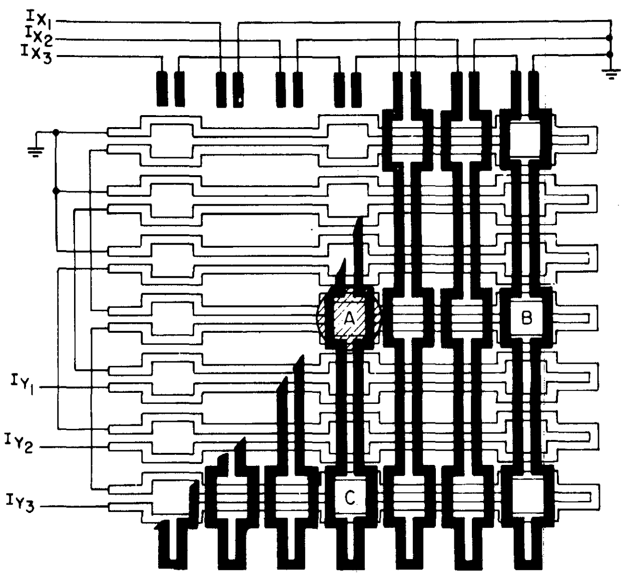


Figure 9—Thin film conductor pattern for two dimensional propagation of cylindrical domains. Conductor dimensions identical with that of Figure 6.

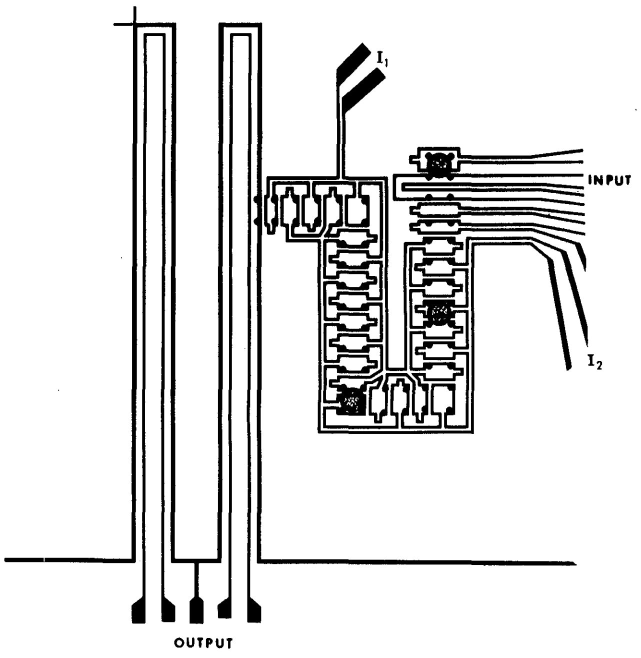


Figure 10—Photograph of the conductor pattern of a undirectional shift register utilizing a biphase propagating source. The circuit contains a controlled replicate input and an output circuit which detects a change in flux. Circuit is capable of propagating domains having a nominal diameter of 4 mils

low as 10 mA have been measured representing drive fields less than 1.0 Oe.

A conductor pattern is shown in Figure 9. Note that the series of loops are interconnected such that there are three separate interleaved circuits. Thus, with a three phase system, a domain at position A can be propagated to C with the sequential application of currents $Iy_1$, $Iy_2$ and $Iy_3$. Two dimensional propagation can be performed by simply aligning two identical circuits orthogonal to each other. The domain at position A can now be propagated to B with currents $Ix_1$, $Ix_2$ and $Ix_3$. Bidirectional propagation merely requires a reversal in the three phase sequence. The domains (bits) are spaced on 10.5 mil centers or every third propagate loop. This is adequate spacing to avoid interactions in materials having a coercive force of 0.25 Oe or higher such as $YbFeO_3$. The resulting packing density is over $6 \times 10^3$ bits/in$^2$.

Figure 10 is a photograph of a unidirectional shift register circuit. The register is equipped with an input and output circuit. Information is written by controlled domain replication and the output circuit detects a change in flux. The circuit is operated with a biphase propagating source. Directionality is achieved with the help of permalloy dots. The dots, which provide low energy sites for the domains, are uniformly shifted with respect to the conductive loops. This asymmetry places the domains in a consistent, preferred position prior to each propagate phase. The permalloy in essence provides a five Oe third phase drive. The permalloy dots are, typically, 4000 Å thick, one mil diameter and spaced on four mil centers along the propagating track. They are deposited on pedestals, fabricated as part of the conductor circuit. This is done to ensure that the permalloy is in intimate contact with the orthoferrite. The biphase register design provides a means of constructing long serial registers without necessitating conductor crossovers. With a biphase system, however, the packing density of domains is about one fourth the propagate positions rather than one third, as in the case of the three phase system. In addition, speed is reduced by virtue of the limit of the pseudo-drive provided by the permalloy.

A suitable material for use with the device is $TmFe0_3$. A platelet two mils thick, exhibiting domains three to five mils in diameter was used. Operation is initiated by placing a "source" domain in the starting loop. To insert a bit, the larger loop encompassing the replication (hairpin-like) conductor is energized, centering the source domain over the replication conductor. After the domain is split, one section is returned to the start position and the other is simultaneously shifted two loop positions to the start of the register. The domain is shifted through the register until it reaches the output circuit. The two outer conductor loops are part of the readout drive circuit while the two inner loops comprise the sense circuit. The readout drive loop nearer the domain is energized drawing the domain into the loop and then expanding it to the extent of the loop. The domain is then collapsed by a reversed drive through both loops. The resulting flux change is detected on one sense loop and the induced voltage due to di/dt is cancelled with the other. The domain is expanded to an area forty times the area of the cylindrical domain and provides an output of 1.0 mV-μsec. A photograph of the output waveform is shown in Figure 11. Notice the bipolar nature of the waveform. The output circuit has been shaped to not only increase the area of the output domain but also to maximize the rate of change of flux linkages during the collapse phase.

The circuit has been operated at speeds in excess of $10^5$ bits/sec using 350 mA propagate currents. The minimum replicate drive pulse is 750 mA, 1 μsec wide. The nominal readout drives for domain expansion and collapse are 530 mA and 700 mA, respectively.

### *"Angelfish" circuits*

We have progressed through three phase conductor circuits where the propagation direction is determined by the sequence in which current pulses are applied and two phase conductor-permalloy circuits where the propagation d'rect'on is built in by a nonsymmetric conductor-permalloy alignment. A logical progression is the possibility of an all permalloy circuit to interact with, and thereby propagate, domains in orthoferrite. There are, in fact, two such general classes of circuits and they will be discussed in this and the following section.

The first class, coined the "angelfish" circuits, utilize the fact that a cylindrical domain can be modulated in size by increasing or decreasing the bias field. Motion is achieved by maneuvering this pulsating cylindrical domain in and out of asymmetrical energy traps. The traps are created by wedge shaped films of high permeability permalloy placed in contact with the orthoferrite platelet.

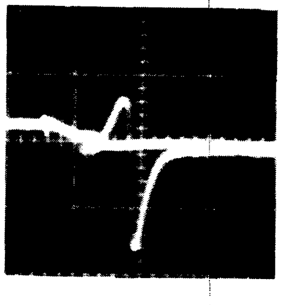


Figure 11—Photograph of the output waveform from circuit shown in Figure 16. Horizontal scale is 1 μsec/div; vertical scale is 2 mV/div

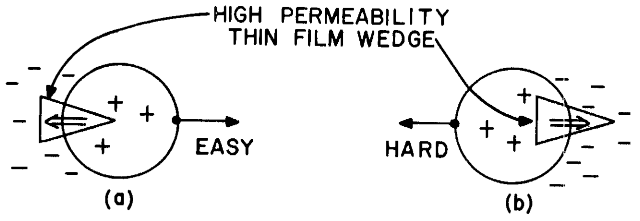


Figure 12—Domain positioned on a wedge-shaped high permeability permalloy thin film. The domain is more easily moved off the point of the wedge (a) than the blunt edge (b)

The interaction which exists between a cylindrical domain and a wedge is illustrated in Figure 12. The domains assume a position on a wedge where the magnetostatic energy is minimized. It was confirmed by experiment that from this position a domain is more easily moved off the point (a) rather than the blunt end (b). The mechanical analogy is that it is easier to walk up a ramp than to scale a wall. A shift register can be built which propagates domains along a series of wedges by means of a periodic modulation of the diameter of the domains. During the expansion phase the leading domain wall reaches out to latch onto the blunt edge of the next wedge and during the contraction phase the trailing domain wall slides off the point of the wedge that held it. This pushing and pulling action provides the unidirectional motion desired.

An experimental 32-step shift register, shown in Figure 13, propagates domains continuously around a circle. The permalloy circuit is photoetched from a 4000 Å permalloy film. The size can be estimated by noting that the outer ring is 50 mils in diameter. The inner and outer permalloy rings provide lateral stability to the domain as it travels. Lateral stability is not required because of any inertia associated with the domain, but to ensure that the domain will expand and contract *along* the direction of motion rather than across. Operation is obtained as the bias field is oscillated between the extremes of 38 to 44 Oe. The orthoferrite used was a 2.3 mil thick platelet of $Tb_{0.5}Tm_{0.5}FeO_3$.

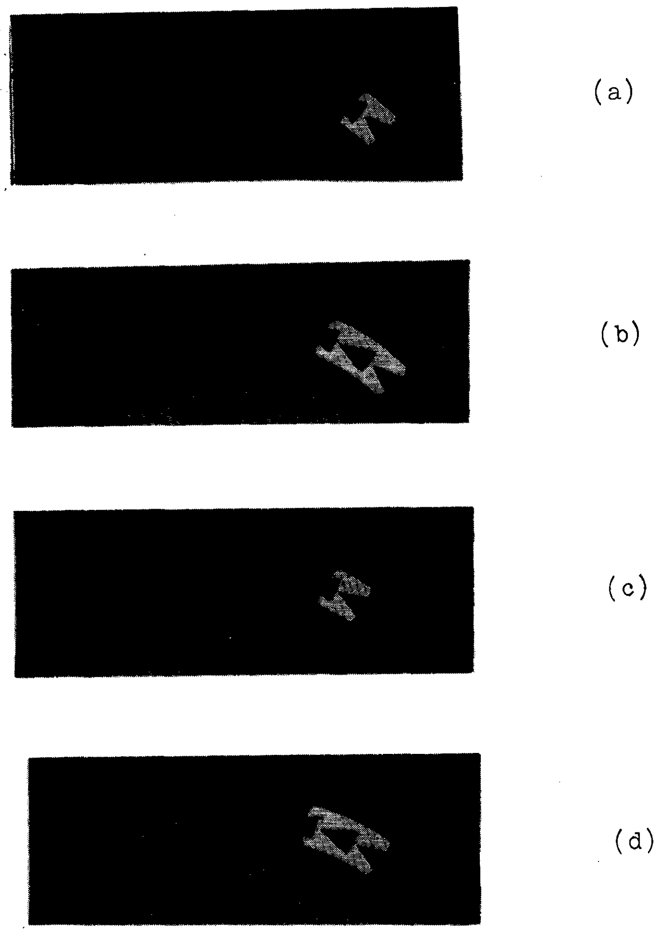


Figure 13—A section of a 32-step unidirectional ring "angelfish" register. The bias field is 38 Oe (a), 44 Oe (b), 38 Oe (c), 44 Oe (d). Motion is counterclockwise

### *Propagation by "T-BAR" permalloy circuits*

In a second method of propagation an in-plane rotating field acting on a structured permalloy pattern generates traveling positive and negative magnetic poles to selectively attract and repel and thereby control the motion of a cylindrical domain. A variety of permalloy patterns are suitable and one such pattern, the T-BAR, is illustrated in Figure 14. The name, T-BAR is, of course, identified with the high permeability thin film permalloy pattern shown in contact with the upper surface of an orthoferrite platelet.

The operation of this circuit will be most readily understood after a study of Figures 14 and 15. First the bias field is adjusted to maintain a stable cylin-

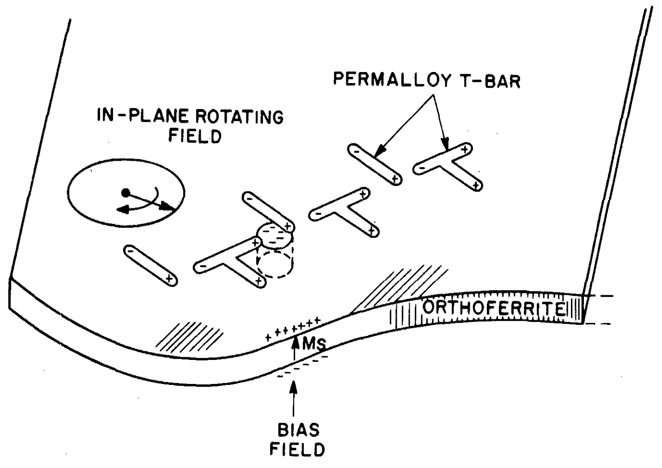


Figure 14—Isometric view of permalloy T-BAR pattern in contact with surface of orthoferrite platelet. Rotating in-plane field generates poles which cause the domain to move

drical domain. Next assume that a field is applied in the plane of the orthoferrite and directed as illustrated in Figure 14. This in-plane field, which has very little direct effect on the orthoferrite, produces magnetic poles in the structured permalloy circuit thereby providing the cylindrical domain with the low energy rest position shown. Clockwise rotation of the in-plane field causes a systematic redistribution of the magnetic poles in the permalloy and the domain responds by moving from left to right as photographed in Figure 15(a)-15(e). With each rotation of the field the domain advances one period of the circuit. The propagation direction may be reversed by rotating the field in the counterclockwise sense.

Figure 16 shows a typical domain generator. The entrance to the T-BAR propagating channel is from the left if the field is rotating clockwise. The large generator disk at the entrance maintains a domain which stays in contact with the + poles formed on the disk by a rotating transverse magnetic field. As the field rotates to the position shown in Figure 16a, the domain is forced to pass over the first + pole formed at the left end of the propagating channel. When the field rotates another quarter cycle, Figure 16b, one end of the domain becomes attached to the advancing + poles of the propagating channel while the other remains attached to the + poles of the disk. As the field rotates further, Figure 16c, the two ends of the domain are forced to travel in opposite directions, and a negative pole distribution begins to build up near the center of the stretching domain, forcing it away from the disk. When the negative pole distribution is maxi-

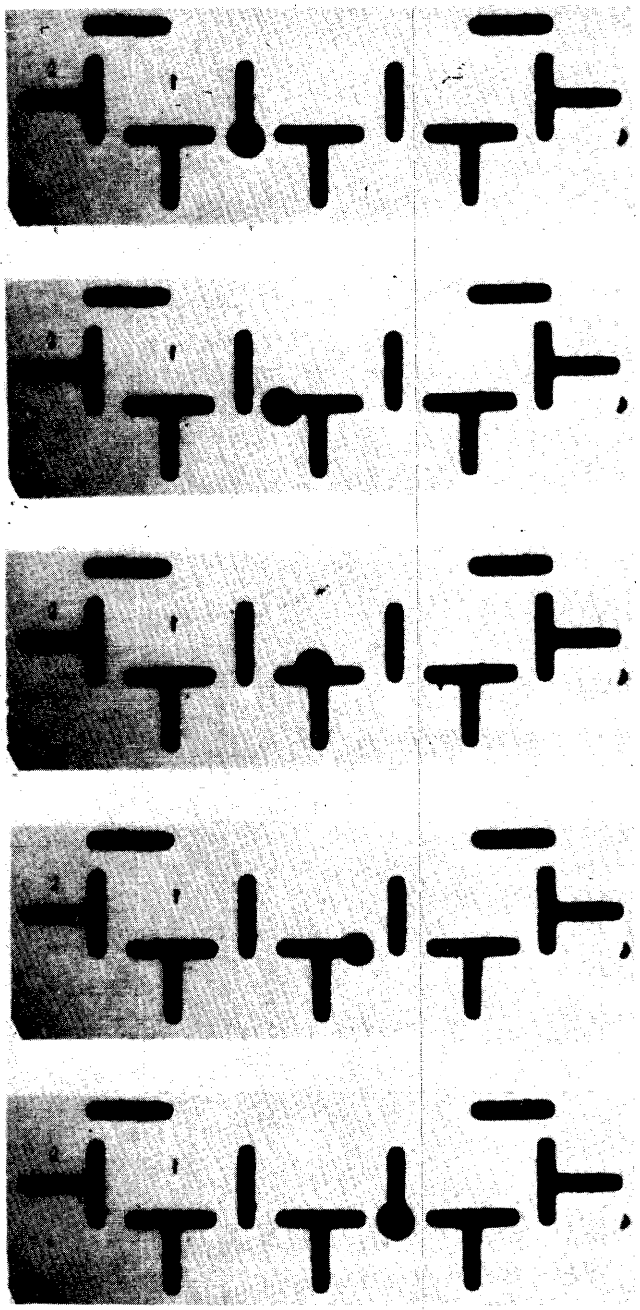


Figure 15—Sequence of photographs showing a 2 mil diameter domain propagating as the field rotates clockwise through 360°

mum near the stretched portion of the domain, Figure 16d, the field from the disk shrinks that portion of the domain width until it becomes unstable and the domain suddenly ruptures into two portions, one remaining on the disk and the other remaining in the propagation channel. Both domains then return to a

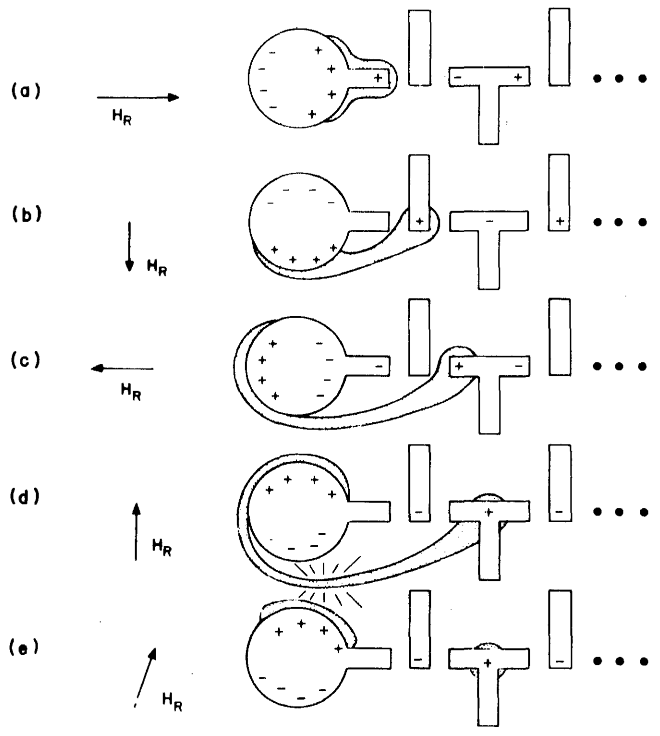


Figure 16—Domain generation—A permanent domain associated with the rotating + pole configuration of the generator disk is forced to stretch when one end becomes trapped in the T-BAR propagate channel. When the in-plane rotating field $H_R$ is directed upward, the − poles near the stretched portion of the domain cause it to sever into two, leaving a newly formed domain in the propagate channel

domain size determined by the bias field with the result shown in Figure 16e.

In general the minimum transverse field required for domain generation is larger than the minimum field for propagation; therefore, insertion of domains into a single channel can be controlled by increasing the amplitude of the rotating transverse field for either an entire cycle or for only that portion of the cycle (approximately ¼ cycle) where the domain becomes stretched to its maximum. Insertion of information in multichannel devices (say up to ten channels) can be controlled by designing the geometry of the generators so that either the amplitude of the rotating field, or the portion of the cycle it must be increased, or both, is different for different channels.

An example of domain generation uses a magnetic overlay made from 8900 Å isotropic permalloy. The T-BAR propagation channel has the same dimensions as previously stated and the generator disk is 9 mils in diameter with a 2.5 mil protrusion into the

TABLE I

| | | | Experimental | | | Calculated | |
|---|---|---|---|---|---|---|---|
| Rare Earth | $4\pi M_s$ | $M_s$ | (mils) 2r | (Oe) field | (mils) Thick, h | (mils) $\ell_d$ | (ergs/c $\sigma_w$ |
| Y | 105 | 8.4 | 3.0 | 33 | 3.0 | 2.5 | 1.8 |
| La | 83 | 6.6 | | Not Available | | | |
| Pr | 71 | 5.7 | | Not Available | | | |
| Nd | 62 | 4.9 | 7.5 | 3.2 | 2.0 | 4.4 | 1.1 |
| Sm | 84 | 6.7 | 6.0 | 3.0 | 1.1 | 2.9 | 1.3 |
| Eu | 83 | 6.6 | 5.5 | 10.5 | 2.0 | 3.7 | 1.6 |
| Gd | 94 | 7.5 | 3.7 | 16 | 2.4 | 2.9 | 1.7 |
| Tb | 137 | 10.9 | 1.7 | 51 | 2.2 | 1.4 | 1.7 |
| Dy | 128 | 10.2 | 2.0 | 32 | 1.6 | 1.7 | 1.8 |
| Ho | 91 | 7.3 | 4.5 | 12 | 2.1 | 3.3 | 1.7 |
| Er | 81 | 6.5 | 6.0 | 8 | 2.0 | 3.9 | 1.6 |
| Tm | 140 | 11.2 | 2.3 | 37 | 2.3 | 1.9 | 2.4 |
| Yb | 143 | 11.4 | 3.8 | 41 | 3.0 | 3.0 | 3.9 |
| Lu | 119 | 9.5 | 7.5 | 10.5 | 2.0 | 4.3 | 3.9 |
| $Sm_{0.6}Er_{0.4}$ | 83 | 6.6 | 1.0 | 33 | 1.8 | 0.80 | 0.35 |
| $Sm_{0.55}Tb_{0.45}$ | 108 | 8.6 | 0.75 | 61 | 2.0 | 0.40 | 0.30 |

$$\ell_d = \frac{\sigma_w}{4M_s^2}$$

propagate channel. The orthoferrite is a 2 mil thick platelet of $Sm_{.55}Tb_{.45}FeO_3$ with $4\pi M_s = 108$ gauss. The bias field is 42 Oe producing approximately 1.5 mil diameter domains. The transverse field amplitude necessary to generate domains is 20 Oe peak while 10 Oe peak is sufficient to propagate domains.

*Domain logic*

Logic can be performed in cylindrical domain devices by utilizing the repelling forces between domains. T-BAR-like overlays are used to transport domains close enough to allow the interactions to occur. An overlay arrangement particularly useful for performing logic functions is that of an idler position into which a domain can be inserted and forced to circulate within a relatively fixed position as the transverse field rotates.

An example of domain logic uses the permalloy overlay of Figure 17. A logic variable N is determined by the presence or absence of a domain circulating in the idler position formed by the four bars which provide the pole positions four, five, six, seven. The input variable X is determined by the presence or absence of a domain in the T-BAR track defined by pole positions ... —3, —2, —1, 1, and two output tracks 3′, 4′, 5′, ... and 7′, 8′, 9′ ... deliver the logic function $X \cdot N$. N is the flip flop function $N = X \cdot (N-1) + \bar{X} \cdot (N-1)$ where (N—1) is the previous state of the flip flop. Poles 2 and 6 are positioned so that if there is a domain on one of the poles, and none on the other, poles 3 or 7, respectively, are preferred

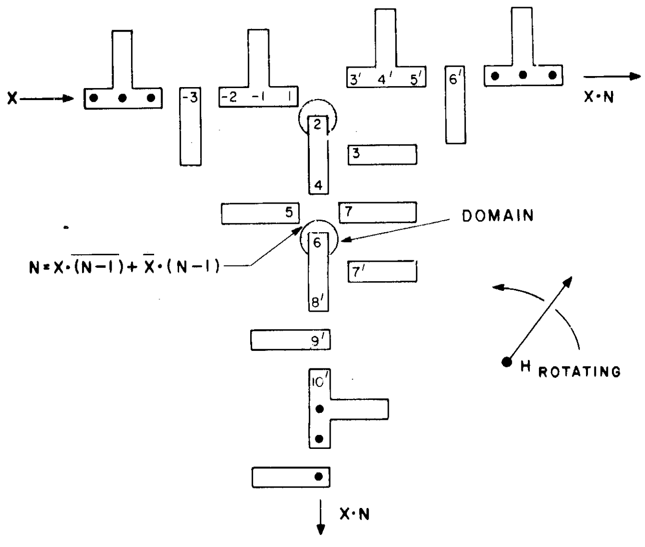


Figure 17—Cylindrical domain flip flop—The state of the flip flop is determined by the presence or absence of a trapped circulating domain at the sequencing pole positions; (idler) 4, 5, 6, 7. Each new domain entering the input channel x changes the state of the flip flop by becoming trapped in the idler if it is full

over poles 3′ or 7′ for the next step. As the transverse field rotates counterclockwise, a domain entering this device will travel along successively generated poles −3, −2, −1 and 1. When it reaches pole 2 it makes a decision to go to pole 3′ or 3 depending on whether a domain is present or not on the idler position 6. If, a domain is present on 6, the two domains repel each other and go to poles 3′ and 7′ when the field rotates the next quarter cycle and henceforth stay on the output tracks 3′, 4′, 5′ ..., and 7′, 8′, 9′ ..., leaving the idler position empty. However, if there is no domain on pole six when the input domain reaches pole two, the input domain goes next to pole 3 and becomes trapped in the successively generated idler poles 4, 5, 6, 7, 4, 5 . . . until a new domain from the input track forces it out. The device, therefore, acts like a flip flop with one input and two identical outputs. The presence or absence of a domain in the idler position determines the state of the flip flop. A binary counter can be made by using one of the outputs as a carry to succeeding stages. Flip flops have been operated by using 11,000 Å permalloy with the overlay design consisting of the usual one mil by five mil rectangles. The orthoferrite was $TbFeO_3$, with a 54 Oe bias producing 3 mil diameter domains. The rotating field peak amplitude was approximately 17 Oe.

## CONCLUSIONS

We have seen that the orthoferrites provide interesting research material for both the theoretician and the experimentalist. Papers covering the wide swathe from materials preparation to device applications have been published. All available orthoferrites have been evaluated as potential domain wall device materials. It was found, for example, that the use of $Sm_{.55}Tb_{.45}FeO_3$ orthoferrite will maximize the storage density since in this compound the smallest domains are found. Stable cylindrical domains 0.5 mil in diameter allow storage densities of $10^5$ bits/in$^2$.

Techniques for propagating domains at data rates in excess of three megabits/sec have been demonstrated using conductor circuits. The upper limit on the data rate for either the "angelfish" or "T-BAR" is yet to be determined although it is expected that the rate for the latter will be in excess of one megabit/sec. Thus we believe that one of the future applications of domain wall devices will be in large capacity shift registers—a solid state disk file.

Although most of the device work presented in this paper concerned the propagation of domains other efforts have pursued the areas of information insertion and detection, and magnetic logic. Magnetic logic is readily implemented using interactions of domains. Therefore, a second application is expected in special purpose memory-logic systems.

Domain wall devices are fabricated using the production techniques pioneered by the semiconductor industry. Thus these devices should be a compatible companion to LSI in future systems. Domain wall devices require few process steps and as such should be manufacturable in high storage capacity units.

## ACKNOWLEDGMENTS

The authors would like to acknowledge the interest and encouragement shown by H. E. D. Scovil in the domain wall device project. J. J. McNicol, R. H. Morrow and R. J. Psonak built and tested many of the circuits. We are all indebted to A. A. Thiele for supplying most of the theoretical background for this work and for allowing us to publish many of his results.

## REFERENCES

1 A H BOBECK
*Properties and device applications of magnetic domains in orthoferrites*
Bell Syst Tech J Vol 46 Oct 1967 1901-1925

2 A H BOBECK
*Properties of cylindrical magnetic domains in orthoferrites*
IEEE Trans on Mag Vol 4 Sept 1968 450

3 A H BOBECK R F FISCHER A J PERNESKI J P REMEIKA L G VAN UITERT
*Application of orthoferrites to domain wall devices*
1969 Intermag Conf April 15-18 1969 Amsterdam

4 D TREVES
*Studies on orthoferrites at the Weizmann Institute of Science*
J Appl Phys Vol 36 March 1965 1033-1039

5 S GELLER
*Crystal structure of gadolinium orthoferrite $GdFeO_3$*
J Chem Phys Vol 24 June 1956 1236-1239

6 C KOOY U ENZ
*Experimental and theoretical study of the domain configuration in thin layers of $BaFe_{12}O_{19}$*
Philips Research Rpt Vol 15 Feb 1960 7-29

7 A A THIELE
*The theory of circular magnetic domains*
To be published

8 A H BOBECK U F GIANOLA
*Magnetic domains*
Science and Technology No 86 Feb 1969

9 A J PERNESKI
*Propagation of cylindrical magnetic domains in orthoferrites*
1969 Intermag Conf April 15-18 1969 Amsterdam Netherlands

10 F C ROSSOL
To be published

# A new integrated magnetic memory

*by* M. BLANCHON and M. CARBONEL

*THOMSON-CSF*
*Laboratoire Central de Recherches*
Essonnes, France

## INTRODUCTION

Very thin permalloy sheets were used by RCA,[12] in 1963, in order to achieve integrated magnetic memories. In 1964, LFE[5] has described an approach to mass memories ($10^7-10^9$ bits) using this material and an integrated wiring. For different reasons, these two projects were abandoned. This paper shows that the two conditions of success are the choice of the shape of the element and the integration process.

First, the shape of the element is discussed and it appears that the toroidal shape is unsuitable for the realization of large integrated memory planes. Unlike the ordinary core, the three-hole element[4] has very broad tolerances on driving currents and on magnetic characteristics of the material. Therefore, the three-hole core was chosen for the integrated memory plane described in the third part of the paper.

Then, the drawbacks of the usual integration processes are underlined and a new, much more reliable method is proposed. A $16 \times 8$ bits and a $32 \times 36$ bits plane were realized using this fabrication process. The characteristics of these memories are exposed in the last part of the paper.

### *Memory device characteristics*

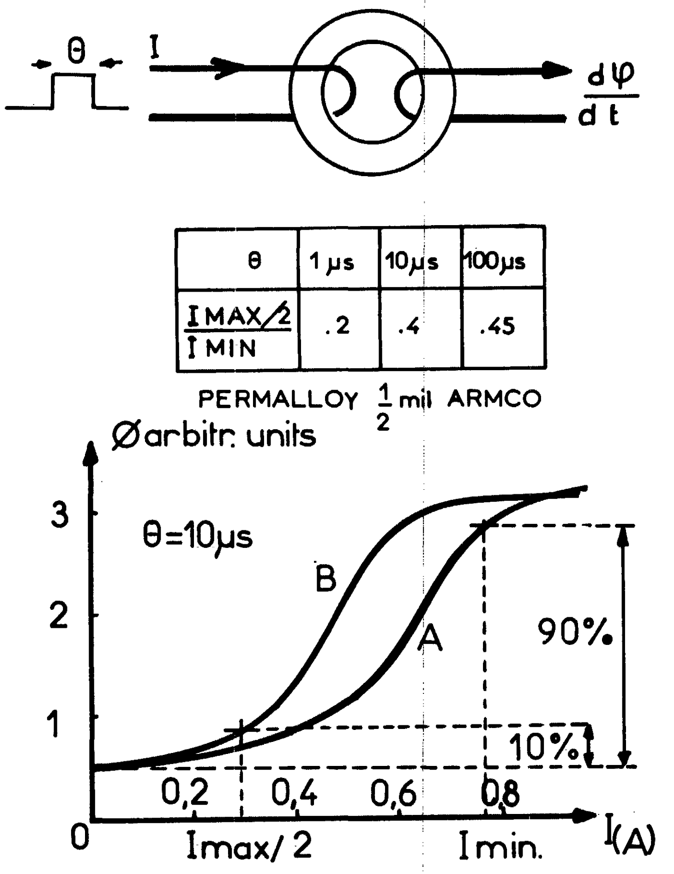
Batch-fabrication of memory planes necessitates a careful study of the characteristics of the memory element. The simplest shape is the toroid.

### Characteristics of the toroid

Consider the element shown in Figure 1 and let us plot the switched flux versus the driving current, when only one current pulse is present (curve A) and when a large number of identical pulses is sent (curve B). For correct memory operation the toroid must switch completely for I and must not switch for I/2. Let us name $I_{MIN}$ the minimum current needed to switch 90 percent of the flux with a single pulse (curve A), and $I_{MAX/2}$ the maximum current allowed to switch only 10 percent of the flux with a large number of pulses (curve B). The required conditions are $I \geq I_{MIN}$ and $I/2 \leq I_{MAX/2}$. This is not possible for the permalloy 1/2 mil toroid where $I_{MAX/2} < 1/2\ (I_{MIN})$. For other pulse widths, other shapes or other thicknesses (1/8 mil to one mil) this is still not possible. Thus, one is then led to use more elaborate driving currents such as bipolar digits or doublet currents.[5] With these improvements, the permalloy toroid memory will work but with relatively tight tolerances. However in batch-fabrication of a large number of toroids, tight tolerances will lead to low yield. Therefore, toroids were abandoned.

### The three-hole element

Permalloy sheet intricate magnetic elements are easily obtained by etching. The three apertured element has many advantages for storage. Diagrams illustrating the operation of the element are shown in Figure 2. The four legs of the element are of equal width. Starting from the clear state, the one-state is written by applying the word-write drive alone. This will work for any value of the word current provided $i_w > i_{wo}$ where $i_{wo}$ is the magnetic threshold of the ele-



| θ | 1 µs | 10 µs | 100 µs |
|---|---|---|---|
| IMAX/2 / I MIN | .2 | .4 | .45 |

Figure 1—S-curves 1/2 mil thick etched permalloy toroids

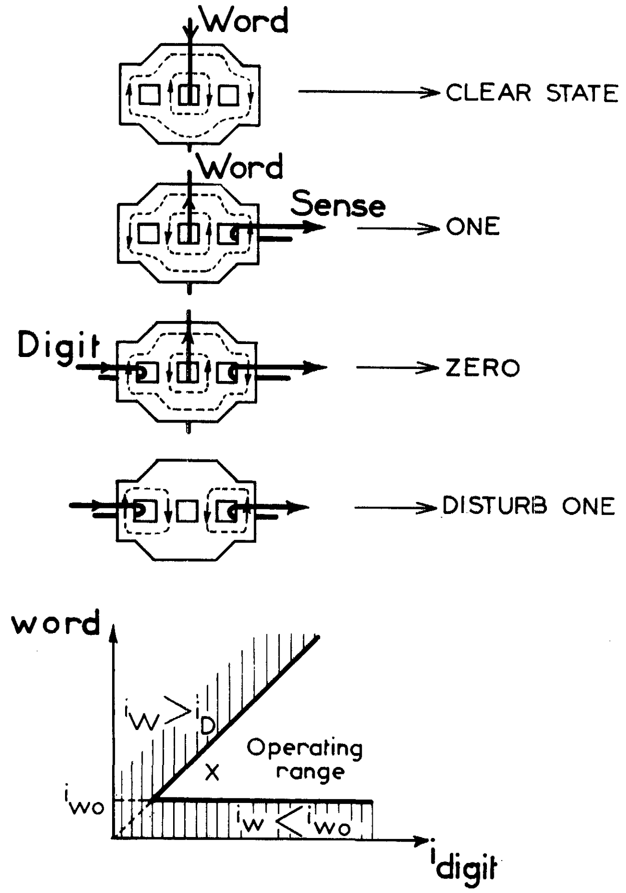


Figure 2—Operation of the three hole element

ment. A zero state is written by applying simultaneously a digit and a word current, the only condition being that the digit current exceed the word current. Applying a disturb digit drive has no effect on the zero but produces a flux rearrangement on the one state, magnetically decoupling the left hole from the output hole. Subsequent disturbs will therefore have no effect. As may be seen from the bottom of Figure 2 the operating range is very wide and is not closely dependent on the magnetic characteristics of the material.

This results in wide tolerances and a wide operating temperature range. Furthermore, a lack or reproducibility in the material or in the element shape is not important. The three apertured element is therefore very suitable for batch processing integrated magnetic memories.

### *Memory plane fabrication*

The processing technique is extremely important for obtaining a good yield. Let us consider an example (Figure 3). The element here is a simple toroid and in the usual integration technique we find a lower winding and an upper winding tied together by a through-connection, thus creating one or two interfaces. These interfaces may be a thin layer of vacuum deposited copper[3] or an electrolytic solder.[6] This results in a serious lack of reliability (broken conductors).

Another drawback comes from the insulation between the winding and the elements. Since there are always pinholes in the insulators, there are often short circuits. One should note that the insulation of the edge of very thin elements is generally extremely difficult.[1,2]

Finally the strains induced by the deposition of the windings may decrease the uniformity of the output signals.

All these drawbacks lower the fabrication yield and the permalloy sheet memories become uneconomical.

The new method described here starts from a permalloy sheet (1/2 mil thick). The permalloy is electroplated with copper (1/2 mil). Using positive photoresist techniques, holes are etched in the plate (Figure

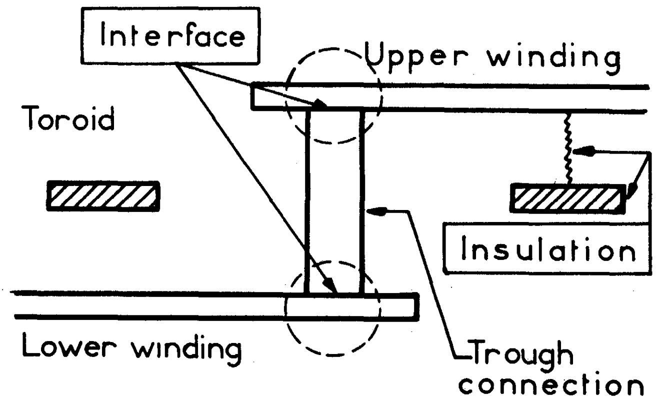


Figure 3—Cross sectional view of an ordinary integrated toroid

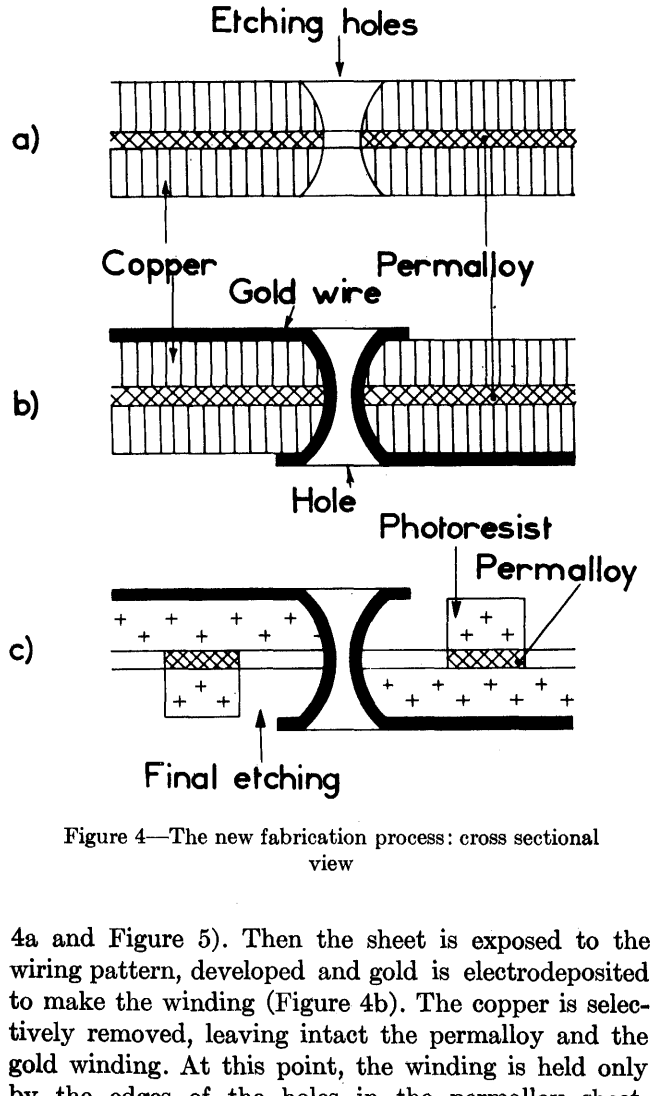


Figure 4—The new fabrication process: cross sectional view

4a and Figure 5). Then the sheet is exposed to the wiring pattern, developed and gold is electrodeposited to make the winding (Figure 4b). The copper is selectively removed, leaving intact the permalloy and the gold winding. At this point, the winding is held only by the edges of the holes in the permalloy sheet,

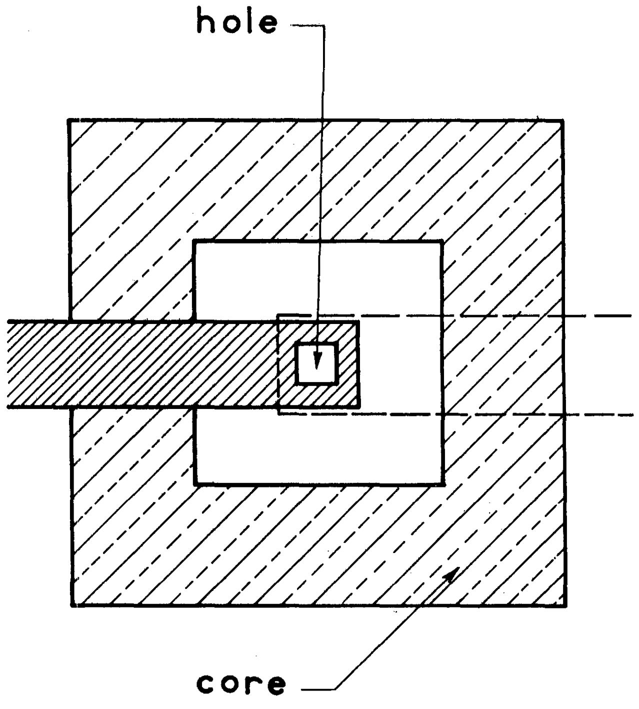


Figure 5—Top view

forming small bridges over the permalloy. The sheet is then dipped in photoresist which takes the place of the copper. After an exposure to the element pattern, the magnetic elements are etched (Figure 4c). If it is desired, the memory may be completed by an encapsulation.

This method is attractive for several reasons:

- Since the upper winding, the through-connection and the lower winding have been deposited at the same time, the wiring is continuous without any interface and this is the reason why it is extremely rare to find a broken conductor.
- Since electroplating tends to fill up all the holes, there are no pinholes at all in a 1/2 mil copper layer. Therefore, there are no short circuits in these memories.
- Since the elements are etched after the wiring, there are no insulation edge problems.
- Mechanical stresses may arise from the electroplating of the copper layer and the gold winding. Removing the copper and etching the element shape relieves the residual stresses of the permalloy.

## *Experimental results*

Memory plane models of 16 words $\times$ 8 bits and

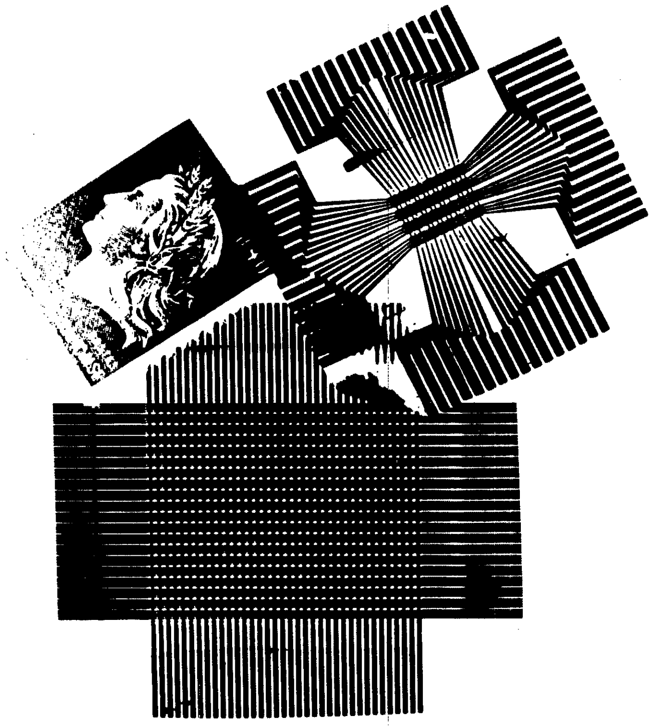


Figure 6—Photographs of 128 and 1152 bits memory planes

32 words × 36 bits were easily fabricated using these techniques (Figure 6). High yield of acceptable planes seems possible even with larger planes.

Characteristics for the 1152 bits storage planes are given in Table I.

*TABLE I—Memory Plane Characteristics*

| | |
|---|---|
| Permalloy thickness | .5 mil |
| Word write current | 50 mA |
| Digit current | 50 mA |
| Read current | 100 mA |
| Density | 250 bits/cm$^2$ (1560 bits/Sqin) |
| Cycle time | $<5$ $\mu$s |
| $V_{out}$ | 1,6 mV ; .7 $\mu$s |

The uniformity of the output signals is excellent as may be seen from Figure 7 where the output of 32 three apertured element are shown superimposed.

## CONCLUSION

Until now, integrated permalloy sheet memories were

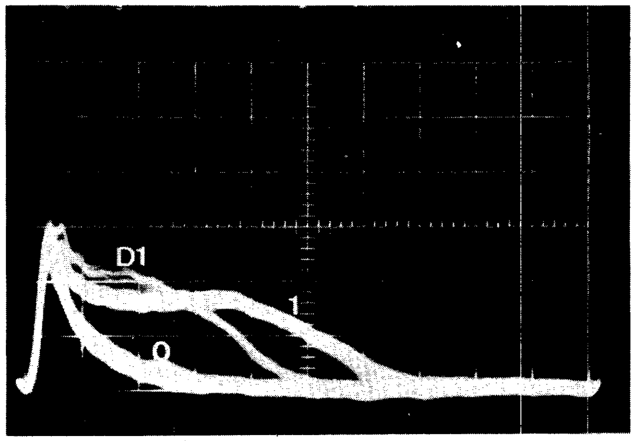


Figure 7—Superimposed outputs of 32 elements (zero. one and disturb one)

Hor 100 ns/cm
Ver 1 mV /cm

not a success. This comes from the choice of the element shape and the processing technique. By using a three apertured element and a new much more reliable fabrication method, these memories seem to have a bright future for mass memories. Higher densities and larger planes (256 × 72) are under study.

## ACKNOWLEDGMENTS

The authors would like to thank J.P. Dupeyron for his assistance on the experiments.

The work reported in this paper was supported by "Direction des Recherches et Moyens d'Essais".

## REFERENCES

1 G R BRIGGS J W TUSKA
*Permalloy sheet transfluxor array memory*
J Appl Phys Suppl Vol 33 No 3 1065-1066 March 1962

2 G R BRIGGS J W TUSKA
*Design and operating characteristics of a high bit density permalloy sheet transfluxor memory stack*
Proc INTERMAG Conf 3-4-1 3-4-8 1963

3 H W FULLER T L McCORMACK C P BATTAREL
*System and fabrication technique for a solid state random access mass memory*
Proc INTERMAG Conf 5-5-1 5-5-4 1964

4 J A BALDWIN JR J L ROGERS
*Inhibited flux—A new operation of the three hole memory core*
J Appl Phys Suppl Vol 30 No 4 58-59 April 1959

5 H CHANG
*Coupled memory elements*
J Appl Phys Vol 38 1203 March 1967

6 M CARBONEL V CHAPTAL
*Batch fabricated integrated all magnetic logic*
IEEE Trans Magnetics Vol MAG 3 535-537 Sept 1967

# Mated film memory—Implementation of a new design and production concept

*by* L. A. PROHOFSKY and D. W. MORGAN

*UNIVAC, Division of Sperry Rand Corp.*
St. Paul, Minnesota

## INTRODUCTION

A high performance computer memory must operate at high speed, require a minimum amount of power, and be capable of operating under extreme environmental conditions. Thin film memories meet these requirements, however, anyone who expected them to become the primary memory technology was certainly premature. Despite its superior performance features, the thin film memory has encountered producibility problems which have prevented it from becoming cost competitive. Univac has developed the MATED FILM* memory concept and a continuous vacuum deposition system which together have overcome previous producibility obstacles and now make the evaporated film memory a serious contender for main store applications.[1] The features which are new and unique to this approach are:

1. Economical continuous deposition for 16-hour periods with all deposition parameters maintained in equilibrium.
2. The closed-flux path design has wide operating margins and provides an exceptionally low susceptibility to process variations.
3. Changing the film array organization from a word-bit matrix to a bit-slice array has greatly reduced the number of connections and process steps required to fabricate the memory stack.

This paper describes: (1) the MATED-FILM memory design which can be adapted to a wide range of capacity and speed; (2) the continuous vacuum deposition facility which has been developed for the production of MATED FILM memories; and (3) a wide temperature, 500 nanosecond, 5 × $10^6$ bit memory which has been built and tested.

*Storage element*

### Construction

The storage element (Figure 1) is formed by a deposit of two layers of nickel-iron separated by a thin, deposited, copper conducting strip. Silicon monoxide layers isolate the nickel-iron layers from the copper layer. The layers of silicon monoxide are sufficiently thin so they do not interfere with the magnetic coupling of the two nickel-iron layers.

Each layer is deposited through masks on glass substrates in a vacuum chamber (Continuous Vacuum Deposition System). When completed, the copper conducting strips form the sense/digit line enclosed by the two magnetic layers.

An etched high permeability keeper is placed in close proximity to the deposited element (Figure 2). The storage element and the keeper are separated by a one mil insulating coating to avoid any shunt current paths through the keeper. The storage element now has a closed magnetic flux path for both the transverse and longitudinal axes. The advantages of this configuration are: (1) The transverse and longitudinal demagnetizing fields are reduced. This results in lower drive currents and improved operating margins. (2) Interaction between adjacent bits is reduced to a negligible

* Trademark of Sperry Rand Corporation.

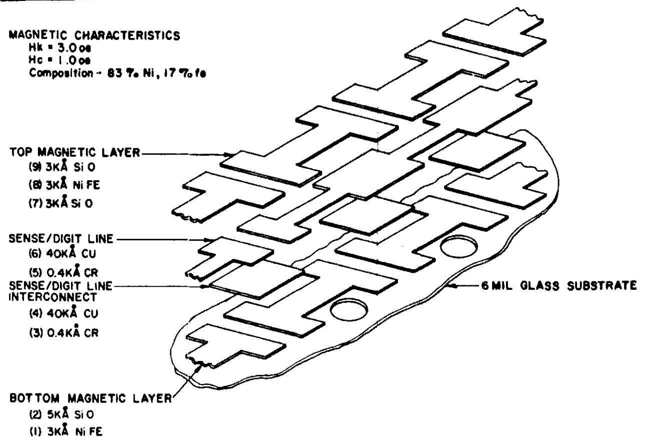


Figure 1—Storage element (exploded view)

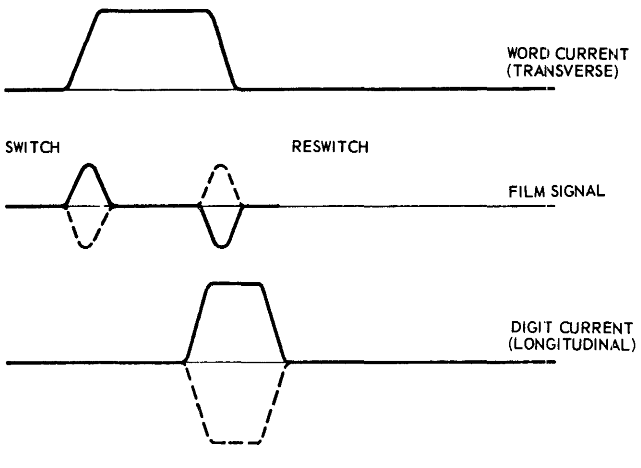


Figure 3—Signal drive current relationship

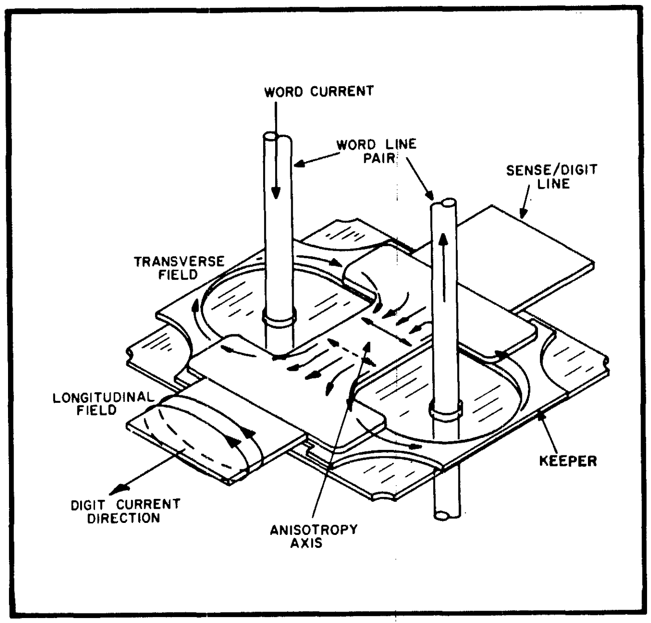


Figure 2—Storage element drive fields

level. (3) Word line to sense line capacitance, which is a source of word noise, is minimized.

## Theory of operation

During deposition, a strong magnetic field produces a uniaxial magnetic anisotropy in the films of the storage element. Therefore, magnetization of the storage element exhibits a preferred axis in the plane of the element normal to the depositeds ense/digit line.[2] A stored "1" or "0" magnetic state of the storage element is determined by the direction of magnetization around the sense/digit line and parallel to the anisotropy axis of the film. The magnetic flux resulting from a stored "1" or "0" closes through the silicon monoxide insulating layers and around the sense/digit line.

Readout of the storage element is accomplished by passing word current through the word line. The resultant transverse field rotates the magnetization of the storage element, which induces a voltage in the sense/digit line.

The initial direction of magnetization determines the polarity of the induced voltage. The relationship of the word current, film signal, and digit current is shown in Figure 3. The rotation of the storage element magnetization occurs during the rise time of the word current.

Passing a current of selected direction through the sense/digit line restores or writes a "1" or "0" in the storage element. The resultant longitudinal field overlaps the trailing edge of the word current field and steers the magnetization to a state determined by the direction of the digit current.

## Nominal operating characteristics

Storage element operating characteristics are obtained by plotting output flux as a function of drive currents for prescribed reading and writing conditions. The total output flux is obtained by integrating the output voltage with respect to time. Since in normal operation digit current is common mode in the sense/digit line pair, all digit currents are given as total array currents. This is twice the single element current.

### Output flux vs read word current

Figure 4 shows the output flux of a typical element as a function of element word current for both the "1" and "0" states.

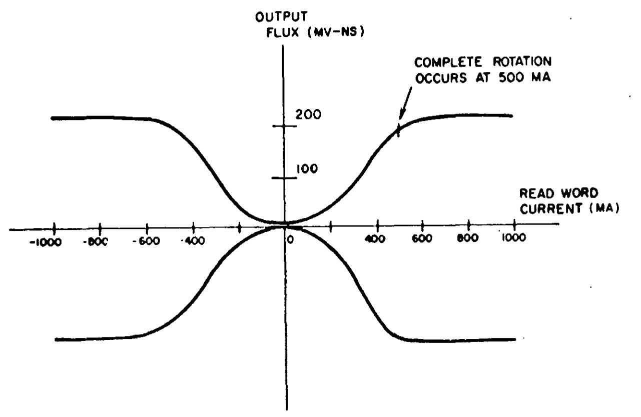


Figure 4—Output flux vs. read word current

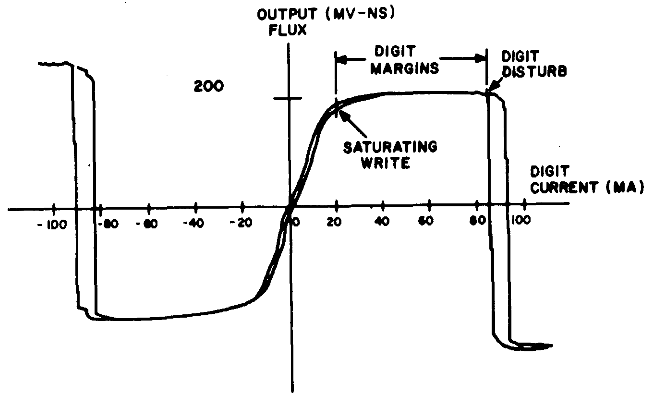


Figure 5—Output flux vs. digit current

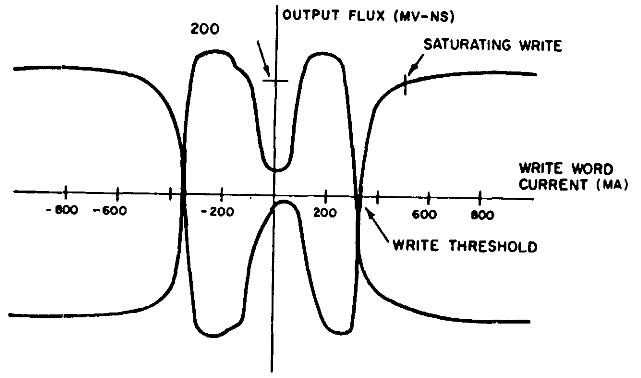


Figure 6—Output flux vs. write word current

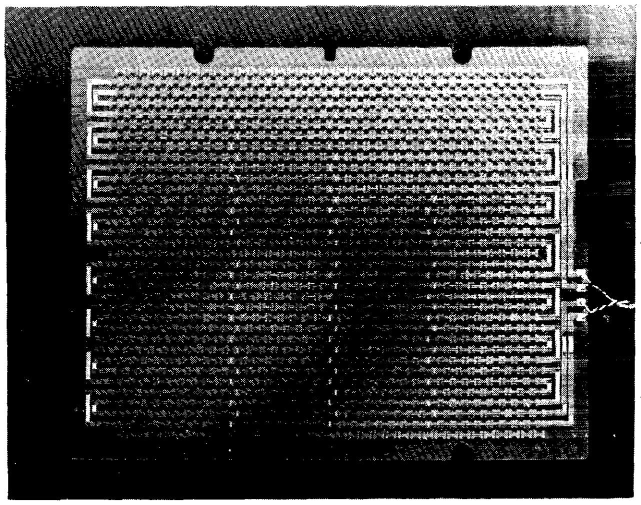


Figure 7—The mated film core array

The curve provides information on element output, symmetry (skew), and operating word current amplitude requirements. The curve is an actual plot obtained by:

1. Writing adverse history 256 times*
2. Writing once in the opposite direction.
3. Reading once and recording flux output at the indicated word current level.
4. Repeating steps (1), (2), and (3), incrementing the read word current each time.
   History and write word current amplitude: 500 milliamperes.
   Digit current amplitude: 50 milliamperes.

* For transverse fields exceeding the write threshold but below the saturating write level, the degree of saturation achieved becomes a function of the number of pulses applied. The first pulse will write a portion of the film while each succeeding pulse writes a little more. In this way, the film asymptotically approaches the maximum magnetized state for the given field.
Adverse history consists of a sufficient number of pulses to ensure that the element is conditioned prior to write with the magnetic state worst case for the write operation. It has been observed that there are no significant history effects beyond 256 pulses. In memory applications, the element is operated beyond the saturating write level, where historv effects are negligible.

## Output flux vs digit current

Figure 5 shows the output flux level obtained with a fixed word current of 500 milliamperes as a function of digit current after repeated digit disturbs.

The plot was obtained by:

1. Writing adverse history 256 times.
2. Writing once in the opposite direction with the indicated digit current.
3. Digit disturbing 256 times with the indicated digit current.
4. Reading once and recording the output flux.

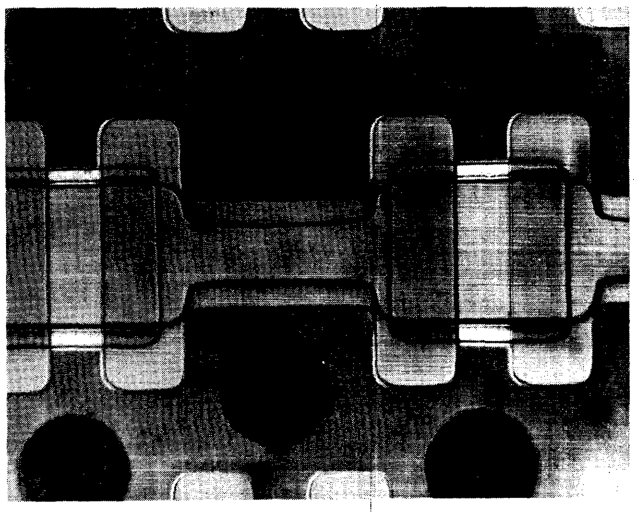


Figure 8—Enlargement of memory array

5. Repeating steps (1), (2), (3), and (4) while incrementing digit current each time.
   Write word current amplitude: 500 milliamperes.
   Write digit current amplitude: 50 milliamperes.

A digit current of 25 milliamperes is sufficient to write, while a current of over 80 milliamperes is required to digit disturb the storage element.

Output flux vs write word current

Figure 6 shows the output flux level as a function of write word current with fixed read word current and fixed write digit current.

The plot was obtained by:

1. Writing adverse history 256 times.
2. Writing once in the opposite direction at the indicated word current.
3. Reading once and recording flux output.
4. Repeating steps (1), (2), and (3) while incrementing the write and history word current each time.
   Read word current amplitude: 500 milliamperes.
   Digit current amplitude: 50 milliamperes.

The write threshold occurs at 300 milliamperes and a saturated write is accomplished at 500 milliamperes.

## Memory array

An array of 1024 active storage element plus 32 spares is vacuum deposited on a photo-etched glass substrate (Figure 7). The deposited sense/digit line pair links all bits on the array making this a 1024 word by one bit slice of the memory. The storage element, in the shape of a capital I, is shown in the enlarged view of the array (Figure 8). The body of the I is the active region of the element. The remainder of the element is always in a demagnetized state; however, it serves the useful function of reducing the transverse demagnetizing field. The two holes, which straddle each element, accommodate the word lines.

*Continuous vacuum deposition system*

The continuous vacuum deposition system is the one most significant feature which sets MATED FILM memory array processing apart from conventional batch processing systems. Operational shakedown tests on the system have been completed. These tests demonstrated the system's feasibility as well as its capability. The capacity of the present system is $10^8$ bits per year. A program to increase this rate will put the facility in a full capacity mode of $1.6 \times 10^9$ bits per year by early 1970.

## Continuous fabrication

MATED FILM memory arrays are fabricated by a continuous vacuum distillation process using an in-line concept of material flow. Glass substrate blanks travel sequentially through four deposition chambers (Figure 9) where progressive layers of magnetic alloy, copper, and insulator material are deposited through precision contact masks. The lost time due to pumpdown, substrate heating, and substrate cooling in a batch process is saved in this continuous process once steady state vapor composition is achieved. Typically this is 20 minutes after start-up.

Within the vacuum chambers, the various materials are vaporized continuously and the deposition is monitored and controlled automatically. A production cycle of 16 hours during a 24-hour period is realized using this process.

In conventional batch distillation processes, the composition of a multi-component vapor is a time dependent function. The higher volatility fraction vaporizes in a proportion greater than its melt fraction. To achieve a deposited alloy film of a precise composition, for example zero-magnetostriction iron-nickel alloy, the vapor stream must be captured at a point in time determined by composition versus distillation time.[3] With continuous fabrication, the process control is built around negative feedback techniques which routinely control the composition of the alloy vapor

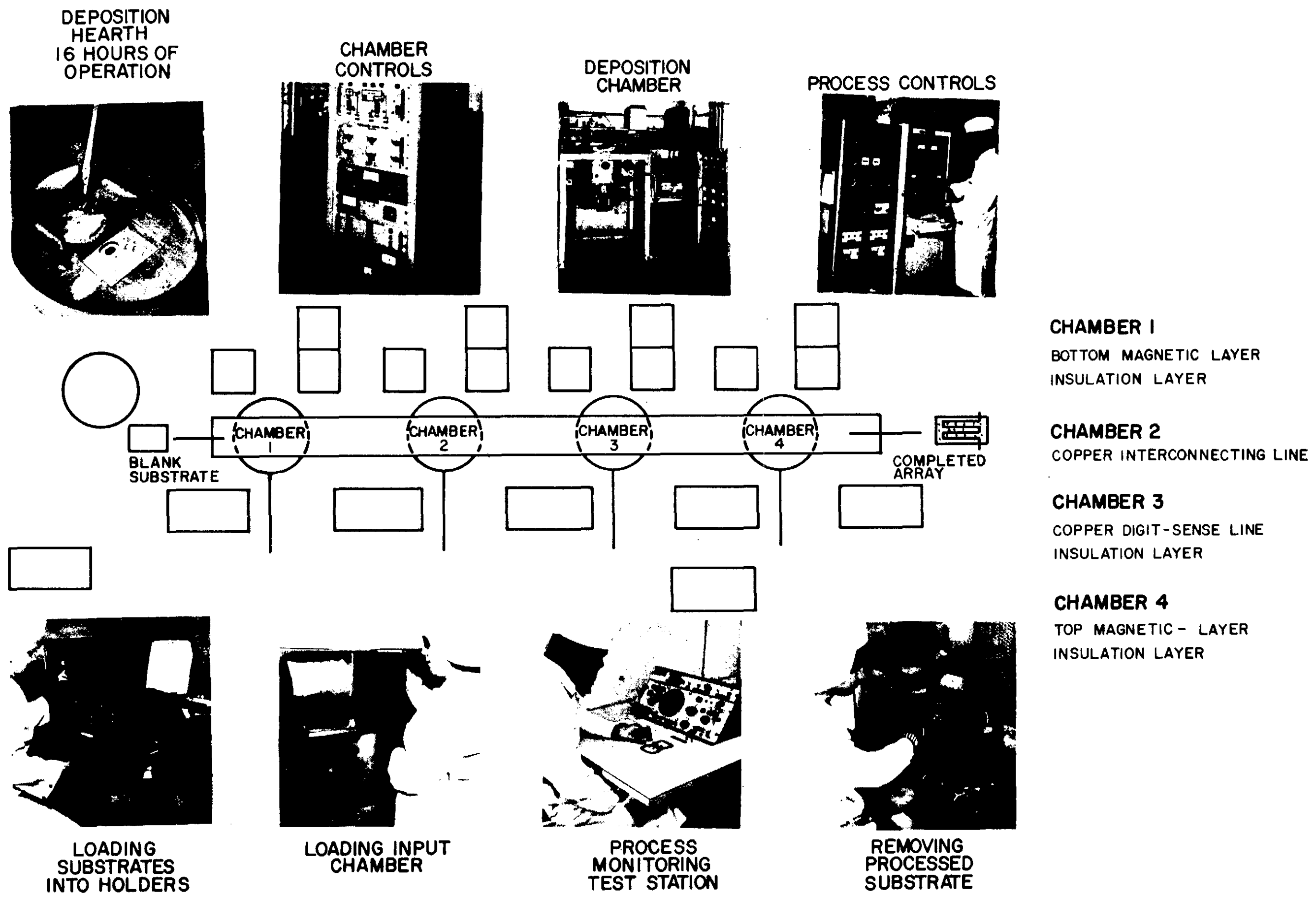


Figure 9—Schematic, thunderbird facility

for continouus periods of 16 hours. The vapor distilled by this steady-state process produces constant zero-magnetostriction nickel-iron vapor for time periods measured in hours rather than minutes.

The separate production stations (the four deposition chambers) of the continuous system permit corrections to be made easily and quickly. Also, the continuous emergence of arrays allows for prompt monitoring of the system. After each deposition stage, the substrates are removed and inspected. As soon as a defect is detected, the continuous system can be stopped and the problem isolated and corrected. Loss of process control in the batch system, no matter when it is detected, usually results in loss of the entire batch.

## System description

MATED FILM memory arrays are fabricated in four identical continuous vacuum evaporators. Each evaporator (Figure 10) consists of a deposition system, a transport system, and pumping system.

The deposition system is capable of evaporating up to three source materials concurrently at specific rates. Electron beam heated sources are used for nickel-iron, copper, and chromium. The SiO source is resis-

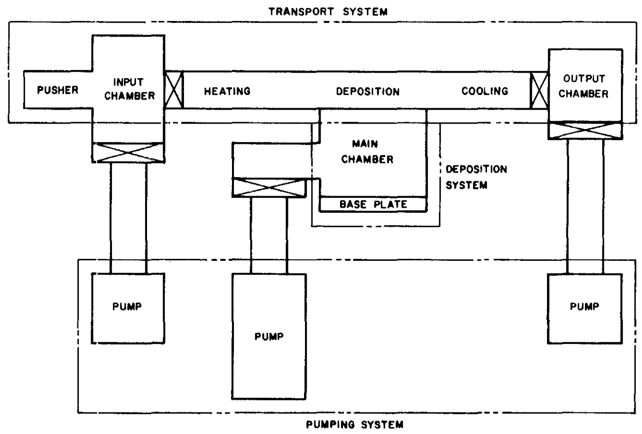


Figure 10—Continuous vacuum evaporator

tance heated. Evaporant shutters above each source automatically expose the substrate for a predetermined time interval. The evaporation rate of the nickel-iron and copper sources is controlled using a vapor rate monitor. The monitor signal is used to regulate the electron beam gun emission current. Evaporated materials are replenished by wire feeders which draw nickel-iron or copper wire from a spool and guide it into the molten source. The removable base plate, which contains all of the deposition equipment except the shutters and vapor rate monitor, fits onto the bottom of the main chamber.

The transport system moves substrates from a magazine in the input chamber to the main chamber, where the depositions are made, and then into the output chamber. Heaters raise the substrate to deposition temperature during transit from the input chamber to the deposition chamber. The substrates pass through a water-cooled tunnel in the cooling section of the transport system, which cools them to handling temperature before they enter the exit chamber.

The automatic pumping system has three interlocked subsystems controlled from a single console. The pumping system maintains high vacuum in the deposition section of the evaporator, while cycling the input and output chambers from atmospheric pressure down to high vacuum as required by the transport section.

## System operation

After the substrate is inspected for possible defects, it is placed in the substrate holder and covered with the first mask. Subsequent substrates and masks are loaded in holders and placed in a cartridge. A cartridge of holders is loaded into the input chamber of Station 1. The holders are automatically ejected from the cartridge and pushed sequentially toward the deposition chamber. Within each deposition chamber the substrate is exposed at two of the three positions or windows available. At the first position the bottom magnetic layer is deposited in the memory bit pattern. At another position the silicon monoxide is deposited over the magnetic alloy through the same mask. When this process is completed, the holders are pushed to the exit chamber.

After it has been removed from the exit chamber, the substrate with the first magnetic alloy and silicon monoxide layers is inspected, and returned to the substrate holder with the mask for the interconnecting elements. The cartridge is then reloaded into the input chamber of Station 2. Using the same procedure, a thin adhesion layer of chromium is deposited for the sense line interconnecting elements, followed by an overlaying deposit of copper.

At Station 3, the substrate is again removed, inspected, and loaded into the input chamber using different masks for the sense/digit conductor deposition. Chromium, copper, and silicon monoxide are deposited using all three exposure positions.

At Station 4, the top magnetic layer and silicon monoxide are deposited. The outer film of silicon monoxide seals and insulates the memory bits.

At this point, the completed film arrays are ready for functional testing before being assembled into memory stacks.

*Memory stack construction*

The MATED FILM memory can be thought of as a two wire system. One axis of stringing and its associated connections are an integral part of the previously described deposition process. To complete the stack it is only necessary to string the word axis and terminate these word lines in the word diode selection matrix.

The memory plane assembly is formed by bonding two film arrays to a single keeper, as shown in Figure 11. In this form the array is less susceptible to scratching or cracking during subsequent assembly.

The film arrays are combined to form a 1024 word by n bit substack with one array for each bit in the memory word. The substack can then be arranged in various series/parallel configurations to meet specific system requirements. The design will accommodate word length up to 256 bits without affecting cycle time.

Figure 12 is an exploded view showing the substack construction. The memory planes are stacked with the etched holes vertically aligned; half of each word loop is connected to the bussed word line header and is threaded down through the substack, while the re-

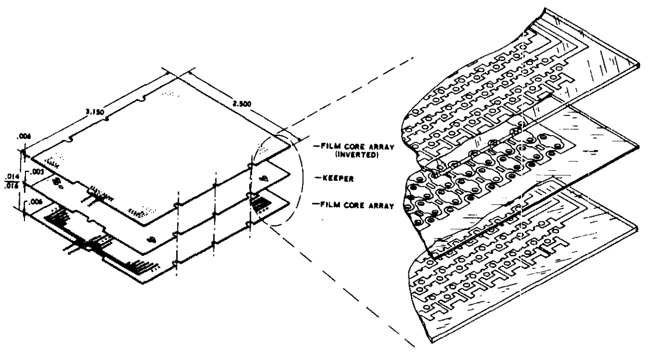


Figure 11—Memory plane assembly

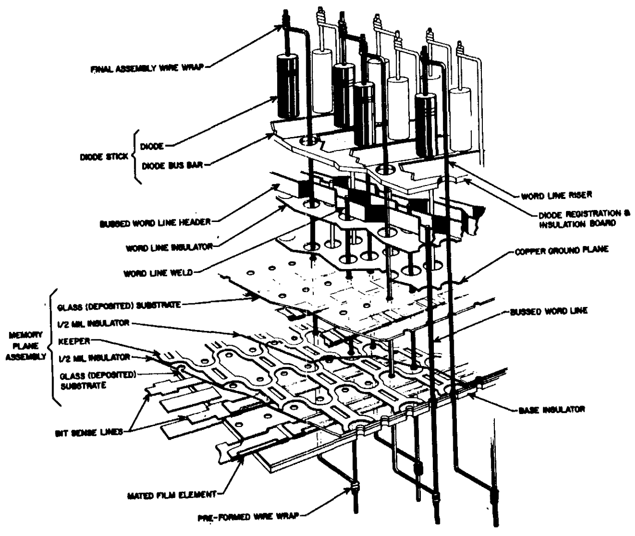


Figure 12—The substack

maining half of each word loop is threaded from the bottom of the substack. The preformed wire wraps connect the word loops at the bottom. The top end of the word loops are wire wrapped to the diode leads. The wire wrap connections are then mass soldered to ensure a reliable electrical connection. The completed substack contains 32 spare words and 10 percent spare planes which are externally accessible. These spare words and planes may be used, without restriction anywhere in the substack. This means that the substack will never require rework unless all of the spare words or spare planes are consumed.

*Memory system*

The memory substack and element design does not vary with the application; however, some of the memory electronics must be tailored to the specific capacity and speed required. One typical configuration which has been built and tested is a 16K word, 32-bit militarized memory with a cycle time of 500 nanoseconds. A sketch of this memory (Figure 13) shows the location of the memory subassemblies.

The heat exchanger which mounts on the front face of the chassis is not shown in this sketch. Cooling is accomplished via thermal conduction from the components to the heat exchanger which is convection cooled by external air.

The stack module (Figure 14) contains a pair of 1024 word, 64-bit substacks mounted on a common plug-in header. The connectors on each side carry the drive lines leading to the diode selection matrix. The stack modules are field interchangeable within and between chassis.

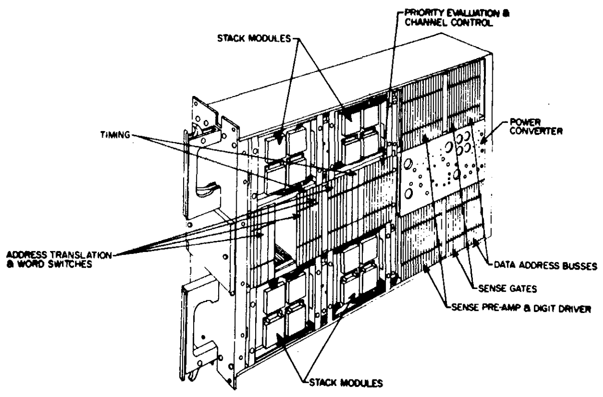

PRIORITY EVALUATION & CHANNEL CONTROL
STACK MODULES
TIMING
POWER CONVERTER
ADDRESS TRANSLATION & WORD SWITCHES
DATA ADDRESS BUSSES
SENSE GATES
SENSE PRE-AMP & DIGIT DRIVER
STACK MODULES

FIGURE13. THE MEMORY CHASSIS

| | | | |
|---|---|---|---|
| WEIGHT | 49 lbs. | CAPACITY | 16K words 32 bits |
| DIMENSIONS | 18.3" x 11.3" x 5.5" | CYCLE TIME | 500 nanoseconds |
| INPUT POWER | 190 Watts | ACCESS TIME | 225 nanoseconds |
| INPUT VOLTAGE | 90 volt (internal dc to ac power converter) | INTERFACE | 8 Channel Asynchronous |
| COOLING | Conduction to a convection heat exchanger | ENVIRONMENT TOLERANCE | Mil-E-16400 Class 1 |

Figure 13—The memory chassis

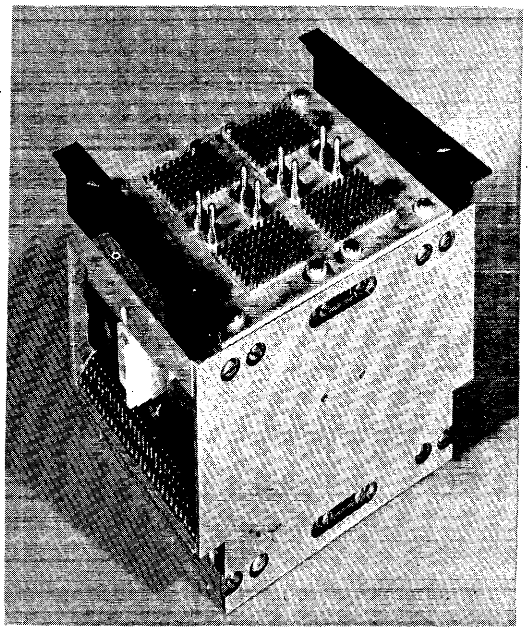


Figure 14—The stack module

## Sense/digit configuration

The total sense/digit line is formed by interconnecting the 1024-bit sections, which are part of the individual substacks. Figure 15 shows one of 64 com-

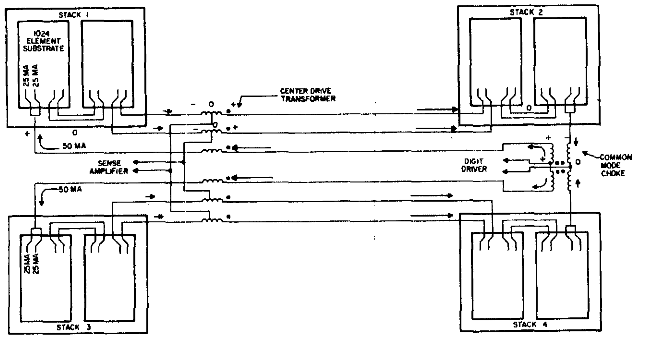


Figure 15—Sense/digit line configuration

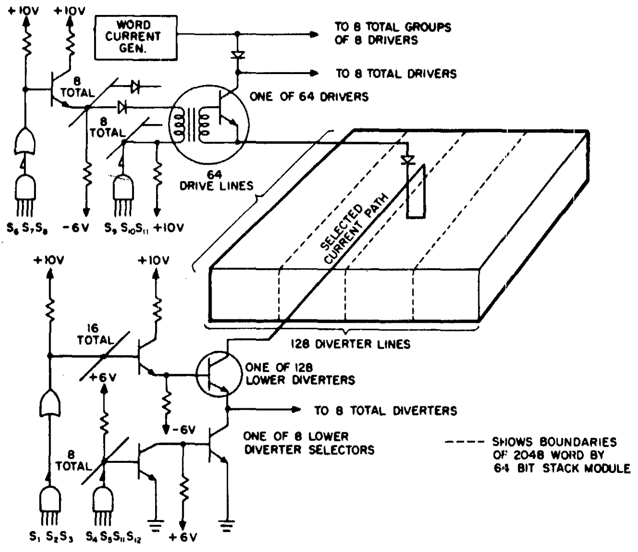


Figure 16—Word selection

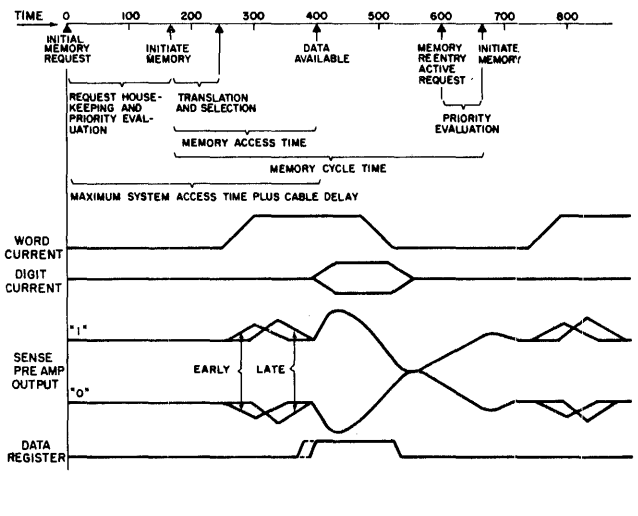


Figure 17—Timing for a typical memory cycle

plete sense/digit lines. Stack modules 1 and 2 form the left and right halves, respectively, of the 4096-bit bridge. Stack modules 3 and 4 form a second bridge and are connected in parallel to a common sense amplifier and digit driver.

The common mode choke ensures that currents flowing in and out of the bridge are equal, and provides both a common mode and differential null at the sense terminals to the degree the legs of the bridge are balanced. This unbalance is controlled so that the digit noise induced into the amplifier is less than three times the signal, a level within the tolerance of the amplifiers. The center driver transformer reduces the time required for the digit current to achieve steady state throughout the line to 40 nanoseconds. Without this transformer, the time would be 80 nanoseconds.

## Word selection

Words are selected by the following method. The four stack modules, each containing 2048 double length words, combine to form the system capacity of 8192 double length words arranged in a 64 by 128 matrix (Figure 16). On the driver side of the matrix, address bits $S_6$, $S_7$, and $S_8$ along with $S_9$, $S_{10}$, and $S_{11}$ are decoded to form an eight by eight matrix which selects one of 64 drivers. Similarly, other address bits are decoded to select one of 16 diverters and one of eight diverter selectors. Word current passes through the word loop which lies at the intersection of the drive line and the diverter line. The word current generator controls the amplitude and timing of the word current pulse.

## Timing

Figure 17 shows the timing for a typical memory cycle. Prior to time zero, all requests were processed and the memory was waiting. Then, at time zero, a memory request arrived at the memory interface. For this condition, 165 nanoseconds are required to acknowledge the request, process it through the priority network and gate the address into the memory address register.

At $t = 235$ nanoseconds, the address is decoded and the proper word and diverter switches have been turned on. Word current is driven through the selected word loop, interrogating the films in that word. The sense signal peaks within the 50 nanosecond rise time of the word current. The polarity of the film signal indicates the stored state. The sense preamplifier output is shown for both a stored "1" and "0."

A sense signal from the near end of the sense line has only a 10 nanosecond delay through the preamplifier; a signal from the far end of the sense line has the additional 40 nanosecond delay of the sense line.

The bottom trace shows the length of time the contents of the data register are valid. During this time, the digit driver is turned on; the polarity of the digit current determines which state is to be stored. On a read cycle, the data is recirculated from the data requester. At this time if the data from the requester is not available, the memory performs a split write cycle while waiting for the data to arrive.

At t = 600 nanoseconds, priority evaluation of active requests begins. If active requests are present, the memory will recycle every 500 nanoseconds.

## Test results

A preproduction model of the memory system described was completed in April 1969 and has been undergoing environmental evaluation. Figure 18 contains "schmoo" data which indicates the threshold of the first bit failure, with the memory system running a comprehensive pattern of writes, reads, and disturbs. Word and digit currents are shown as a percentage of deviation from nominal, $I_w = 700$ mA, $I_d = 45$ mA. The center square represents the system's drive current limits; these limits are $\pm$ 5 percent. This is safely within the usable operating region, as indicated by the "schmoos", for ambient temperature ranges of $-55°C$ to 65°C. The degree of overlap of the high and low temperature "schmoos" eliminates the need for drive current temperature compensation.

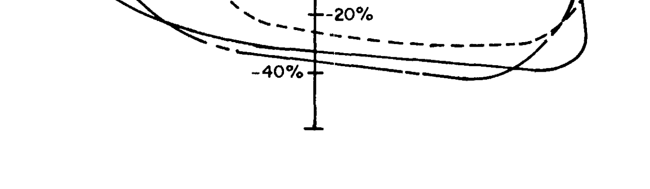


Figure 18—Memory system operating margins

These results show the nominal characteristics of the storage element to be quite representative of the entire memory. The results also show that there are no noise or signal interaction conditions in the stack or electronics that will compromise system margins.

Above the maximum digit current failure is the disturb of unselected bits. This limit approaches the Hc of the films since the element design very effectively minimizes transverse fields on these bits. This would otherwise aggravate the condition. Film dispersion and skew determine the minimum limit of digit current for an adequate write. Word current could not be varied above 20 percent of nominal so the schmoo in this region is not know. Minimum word current failure is caused by the reduced effective rise time resulting in a delayed and reduced signal peak.

## CONCLUSIONS

The existing MATED FILM memory design is conservative, yet competitive. As with any new technology, future development can be expected to enhance performance and reduce costs. The two most significant growth areas for MATED FILM are higher speed and higher bit density. The feasibility of a 200 nanosecond cycle time for systems up to $10^6$ bits has been demonstrated by several partially populated breadboards. Expansion of memory in the word direction has little effect on cycle time. The high bit density in this direction minimizes delay and loading effects.

Part of the future plan for this memory is to double the bit density on the present size array so that each array will contain 2048 bits. This will provide such direct improvements as reduced costs, increased production capacity, and smaller physical size.

## REFERENCES

1 W M OVERN
*Status of planar film memory*
IEEE Trans on Magnetics Vol 4 No 3 Sept 1968 308-312

2 N S PRYWES Editor W CHOW A CHYNOWETH H EDWARDS M HINES D LEENOV V NEWHOUSE A POHM N PRYWES S RUBENS
*Amplifier and memory devices: With films and diodes*
McGraw-Hill Book Co 1965 Chapters 12 13

3 N S PRYWES Editor W CHOW A CHYNOWETH H EDWARDS M HINES D LEENOV V NEWHOUSE A POHM N PRYWES S RUBENS
*Amplifier and memory devices: With films and diodes*
McGraw-Hill Book Co 1965 Chapter 16

# A computer engineering laboratory

*by* D. M. ROBINSON

*University of Delaware*
Newark, Delaware

## INTRODUCTION

The advent of modern electronic computers has expanded the scope of nearly all areas of scientific endeavor. The electrical engineer is perhaps most acutely affected by this expanision by virtue of his two-fold interest in computer processes. He is, as are his colleagues of other scientific disciplines, excited by the computing capabilities now at his disposal. Even more, he is deeply involved by virtue of his responsibility for the conception and design of the computer and its hardware adaptation to a variety of applications. It is to the second phase of the electrical engineer's involvement with computers that our educational activities are directed, that is, to his involvement in the realization of computers or computer-like systems.

### *The environment*

In order to adequately portray this educational activity, it is necessary to describe the environment in which it takes place. This environment will be described as it applies to electrical engineering students at the University of Delaware. However, this is not an atypical situation and the description could apply to many of our universities.

## Present status

Our senior students are now beginning to come from a generation which has grown with the computer. Some have started their association with computing machines in high school or even earlier. All have been through some sort of a problem-oriented first course which leads to machine solutions employing a language like FORTRAN. All have become familiar with the power of the computer for problem solving as early as their first course in Linear Circuit Theory (a candid admission here is that some problems at this level are indeed a bit forced). By the time these students have become juniors, they are aware of user-oriented packages such as ECAP (Electronic Circuit Analysis Program, an IBM applications program) and have employed this type of program in analysis of active and passive networks. Modeling and simulation have become familiar terms and tools to these students.

Except in the very earliest courses, machine computation is not introduced artificially. The students have been challenged by the problems. Courses have not been modified to simply introduce computational techniques; rather, the problem areas have no longer been artificially compressed to exclude the large system or the nonlinear problem which motivates the computational techniques. It should be mentioned that closed-form solutions and functional relationships are sought first. We do not seek to relegate all problems to computer solutions but rather to find a reasonable balance between this and the more traditional treatment of problems.

All of these activities are motivated by the search for solutions to generally traditional problems in electrical engineering; these activities have been termed applications oriented. For the most part, engineering educators tend to center their computer related activities about the capability of machines for solving traditional problems and the vehicle by which this computational power may be focused on their particular discipline. In such application areas, our educational system seems to be responsive to the student's requirements.

## Changes

The electrical engineer's environment is dynamic. An educational system which was responsive to the needs of the past may not now serve. There are new problems of importance, problems which have been spawned by the very existence of the computer. Recent electrical engineering graduates are concerned with the design of systems which may involve a general-purpose digital computer in an on-line control function, a data-retrieval and signal-processing operation or some similar real-time application. Control, communication, pattern recognition, filtering, and numerous other system functions are frequently developed about special-purpose digital computers. As a class, such systems certainly represent a significant portion of today's electrical engineering effort. With these problems for motivation, electrical engineering students view a casual user relationship with computers as simply not being relevant to their educational needs. Their interests and future responsibilities can only be served by an involvement which gives them an intimate experience with this developing environment.

The importance of this changing situation has been recognized at the University of Delaware and over the past five years, several curriculum modifications have been made to strengthen and update our related activities. The subjects which have the strongest relation to this area and, as such, the ones which have received the greatest attention in our revisions, cover such topics as logical design, switching theory and computer organization. The curriculum modifications have extended into such traditional courses as electronic circuits, control systems, communication systems, and information theory. These courses have been modified to emphasize the role of discontinuous elements or discrete systems or to introduce the notion of digital processes. Some course work is immediately related to digital systems and their design while more remotely related course work simply encourages thinking in terms of digital problem solutions.

## Role of the laboratory

These curricular innovations have permitted the development of the general analysis, synthesis or design techniques required for the examination of digital systems. Mathematical descriptions of the situation are developed from models of these systems. As in any physical situation, the conclusions drawn from manipulation of the mathematical models are no better than the original representation of the system; in addition, the modeling process itself is often tempered by the degree of rigor which may be mathematically tractable. Consequently, the conclusions drawn from analysis of the models may fail to give a complete or accurate representation of the physical digital system's behavior. In this area then, as in all areas of engineering, it is felt that laboratory experience acts as a medium through which the reality of the physical situation may be brought to the student. He is made aware of the limitations of his system models and the implications of his modeling process. It is in the laboratory that a student must pursue the details of the subject; this is where he "puts it all together." Thus, progress in the discipline area requires progress in related laboratory experiences.

Enhancing the quality of laboratory studies in digital systems is a process which is not accomplished without assiduous attention. This is true of laboratory studies in general and it is especially the case for a digital systems laboratory. This is at least partially due to the plague which has been termed the "tyranny of numbers." A common characteristic of digital systems is certainly that large numbers of elements are required and that large numbers of connections must be established. Only trivial problems can be attempted in an afternoon spent in the laboratory. Even trivial systems can quickly spread into a maze if usual breadboard techniques are used. Laboratory budgets can rapidly become unrealistic if even only one or two students wish to retain a problem of moderate complexity. Some early efforts were made to develop small patching stations and arrangements which would help alleviate these problems. These efforts served some pedagogical purpose; however, their limited versatility and the relatively slow expansion process did not permit them to foster the desired growth of this area.

The state of our laboratory has been enhanced by the acquisition of a small digital computer and the introduction of this machine into a system which approximates a generalized interface. This system permits physical access to all of the essential computer functions and incorporates facilities for patching connections to external digital logic-modules so that an extension of the computer or an interfacing system may be rapidly established. We have dubbed the system with the acronym DADEC (Design and Demonstration Electronic Computer). This system, which represents only a modest investment, has proved to be a boon in the inspiration of interest and stimulation of growth in this study area.

Several laboratory experiments and exercises have been developed about this DADEC system Some of these are extremely simple exercises which serve to establish familiarity with the machine, its coding, logic levels, etc. Some experiments are rather sophisticated real-time data processing adventures. The set of experiments was designed to support course work from sophomore computer science level through electrical engineering senior projects.

In this paper, the DADEC system will be described and several example problems outlined. The examples have been chosen to illustrate the range of educational levels which may be served using the experimental system, the versatility of the system, an example from several of the particular related course areas, and some problems which may be of general interest.

### *The DADEC system*

The DADEC system is conceptually and practically very simple; a block diagram of the system is shown in Figure 1. Central to the system is a small general-purpose digital computer. A number of digital logic-modules (flip-flops, gates, one-shots, line drivers, etc.) are mounted in adjacent frames with a patch panel which permits the rapid establishment of interconnections between these peripheral elements and the computer. All of the computer interfacing lines are available at terminals on this patch panel.

The majority of the logical building blocks are completely unspecified, that is, any available logic module may be substituted in the patching arrangement. It has been found that a few specific functions are repeated in a great many interfacing problems, and these functions have therefore been prewired on

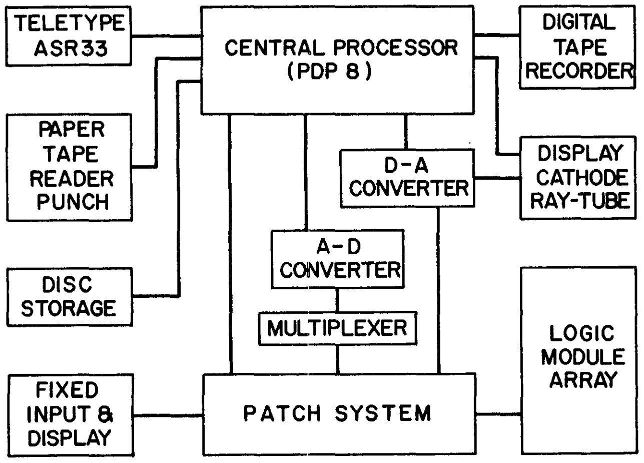


Figure 1—DADEC system —Block diagram

the patch panel (two binary up-counters and one binary up-down-counter). Switch-registers, light-registers, some momentary contact switches and free indicator lights are available as a portion of this generalized interface. Trunk lines are available for connection to remote equipment such as analog tape transports, signal sources, etc.

An analog-to-digital converter is included in this system. Students have designed, built and added a four-channel analog multiplexer. Students have also designed, built and added ten channels of digital-to-analog conversion. A portion of this D-A converter is used to drive a storage oscilloscope facility. This system is by no means static; we are presently adding additional equipment racks for the inclusion of micro-logic modules. Plans include the addition of a paper-tape reader-punch and a disc to the system. An incremental digital tape recorder for accumulation of data for later off-line processing is to be interfaced by the students and added to the system.

A few comments are in order regarding the selection of the particular computer for use in the DADEC system. While the computer is general-purpose, it is not subject to the same set of constraints which govern the selection of a machine for a user oriented computing center. For our purposes, the most important criterion for evaluating a machine is its ability to contribute to the educational process. In order to contribute, it need not have a tremendous core storage capacity or a rapid thru-put capability. Since the machine has been in use, its applications have been concerned with interface problems or the demonstration of system functions and not with its use simply as a computational device. The machine need not have a long word length; there is very little pedagogy which is served by a twenty-four bit machine which is not adequately served less expensively by a twelve-bit machine. Indeed, the short word length and the resulting abbreviated instruction list and core paging system actually serves our instructional purposes. The computer should be easy to interface and adaptable to a large variety of peripheral equipment. It should have inherent compatibility with a family of logic circuits which are readily available. The machine should be easy to service; frequent failures of the system are observed, since many of the experiments involve hardware entry into the internal operation of the machine. Finally, it is a desirable attribute if the machine has at least a limited FORTRAN language compatibility. This enables inexperienced coders to immediately use the system once any additional software is established for addressing peripheral devices.

One currently has a rather large selection of machines which meet these objectives (at least 25 such machines). At the time our decision was made, the list was not so extensive, but we have found that the Digital Equipment Corporation's PDP-8 is a very satisfactory, moderately priced machine.

*Example experiments*

Several example experiments will be outlined in this section. Some of the experiments are, of course, prompted by the requirement that students must first be introduced to this system; however, the predominant motivation is problem solving. When the system was first conceived, the faculty felt responsible for specification of a number of problems to be implemented. We felt that we would be hard pressed to find a sufficient number of examples to insure full utilization of the system, however, the students have been encouraged to suggest problems and their exuberance now prevails. We encourage the students to seek problems from other departments on campus and their suggestions have covered the gamut from exotic time-sharing activities to automatic control of oyster reproduction. These following few examples were chosen from student suggested projects.

### An introduction to the system

The Electrical Engineering Department is responsible for the instruction of computer science majors of the College of Arts and Science in a course that is oriented toward the hardware and architecture of computing systems. For the most part, these students will have had no experience with a digital computer at a more intimate language level than FORTRAN. We find that a simple machine-language program tracing experiment is extremely effective in establishing both an introduction to the DADEC system and the operation of a compiled language. A simple type-out routine is coded in FORTRAN; this program is compiled and loaded along with the operating system. The routine is then executed in a single-step machine-language mode so that all of the required steps of masking, code conversion, communication with a peripheral device, etc., may be examined using the register information supplied by the DADEC system. This experiment is, of course, extremely simple; however, it does illustrate the fact that this somewhat generalized digital system finds use even at early instructional levels.

### An extension of the computer

These computer science students soon become moderately proficient at programming in the assembly language of this machine. Programming instruction is not a part of the course per se; but the relation between "hardware" and "software" which is discussed, quite often naturally brings up coding problems. Near the end of the course they are capable of more ambitious experiments in which additional commands are added to the repertoire of the computer. An example of this is the addition of a "hardware" EXCLUSIVE-OR command. In this experiment, a program controlled input/output transfer is initiated to transfer the contents of two memory locations to external registers. The peripheral portion of the system performs the EXCLUSIVE-OR operation and transfers the data back into the accumulator. Now, of course, a programmer can accomplish a similar result with a sub-routine of some fifteen or so statements. The student is thus faced with an example of what is often called the "hardware-software" trade-off.

### Automatic testing

Within the electrical engineering curriculum, emphasis is placed on designing the class of electronic circuitry which is usually involved in computers. Each student is assigned the problem of accomplishing a "worst-case" design of a discrete element NAND/NOR gate. This design requires that a certain fan-in, fan-out requirement be met at room temperature with any transistor from a given distribution. The DADEC system is used in the evaluation of the students design, that is, in testing of the circuits. The students go through the procedures of design computation, breadboarding, testing, reevaluation of their design, and finally, fabrication of their design on a printed wiring board which is acceptable in the DADEC interface system. The system then exercises their circuit by connecting output loads and applying worst-case signals while circuit conditions are tested with the analog-to-digital converter. The computer gives the student a grade on the lab experiment which indicates how well he met the design objectives.

### Encoding and decoding

A course discipline area is developed in the theory of simple sequential systems. As an example problem, and one which draws upon the student's information theory background, a single error correction digital transmission system is designed. An asynchronous, sequential coder and decoder are realized using NAND gates. This sub-system is patched into the DADEC interface and the computer is used to generate code groups which are transmitted to and received from the transmission system. A random error generator (a computer subroutine) creates a noisy channel or errors in the transmission path. The computer further analyzes the transmission and reports the performance statistics of the system.

### Understanding the computer functions

The particular computer employed in this system has two rapid input-output data transfer mechanisms. These are called single-cycle and three-cycle data-break transfers. These are rather difficult mechanisms for the students to assimilate. This is not because they are conceptually difficult but because of the large number of signals which must be recognized and carefully timed. A simple experiment serves to illustrate both of these data-break facilities. We call this experiment a hardware clear core. In this interface, the single cycle data-break is first called to set zeros into core location zero and one into location one. The three cycle data-break is then initiated with a word count register as location zero accompanied by presentation of all zeros on the data lines. This has the net effect of clearing all core locations except zero and one. The single cycle data-break is then again called to clear these two locations. This is all accomplished with a sequenced switch operation in the interface. While the interface is particularly simple, the experiment does require a sophisticated understanding of the operations of the computer.

#### *Some more challenging experiments*

Student projects are being executed using the DADEC system. In this project environment, rather comprehensive problem areas are either suggested to the students or suggested by the students. They may then pursue a solution of the problem for one or perhaps two terms of their senior year. Several of these problems will be described in greater detail than have the previous problems, since these serve to illustrate the student's approach to problem solving.

### A pulse-height analyzer[1]

The analysis of pulse-height information is quite suitable for digital sub-system solution. This particular pulse-height analyzer is unique in that the pulses are of only about 30 nanoseconds duration and the counting interval must be short (about 50 microseconds) with no dead time between successive count intervals. The student approached the problem by designing an asynchronous sequential circuit which transmits a standardized pulse whenever its input pulses meet the proper amplitude criterion. A description of this system is shown in Figure 2. Two comparators are used as decision elements to determine if the input signal has passed either the low threshold voltage ($V_L$) or the high threshold voltage ($V_H$). The results of these decisions i.e., the output of the comparators, are described by Boolean variables H and L. A flow table which summarizes the required circuit action for any input sequence is shown in Figure 2 (note that flow tables of this type are described in references such as Maley[7]).

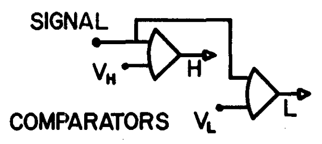
This flow table may successfully be assigned internal state variables ($f_1$ and $f_2$) as shown. The excitation table may be formed, and from these tables excitation functions ($F_1$ and $F_2$) and the output function (Z) may be derived.

| $f_1 f_2$ \ HL | 00 | 01 | 10 | 11 | Z |
|---|---|---|---|---|---|
| 00 | ① | 2 | 4 | 5 | 1 |
| 01 | 3 | ② | 4 | 5 | 1 |
| 11 | ③ | 2 | 4 | 5 | 0 |
| 10 | 1 | ⑥ | ④ | ⑤ | 1 |

FLOW TABLE

| $f_1 f_2$ \ HL | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 00 | 00 | 01 | 10 | 10 |
| 01 | 11 | 01 | 10 | 10 |
| 11 | 11 | 01 | 10 | 10 |
| 10 | 00 | 10 | 10 | 10 |

EXCITATION TABLE



$$F_1 = H + L'f_2 + Lf_1f_2'$$
$$F_2 = H'f_2 + H'Lf_1'$$
$$Z = (f_1 f_2)'$$

FUNCTIONS

Figure 2—Pulse-height detector — Description

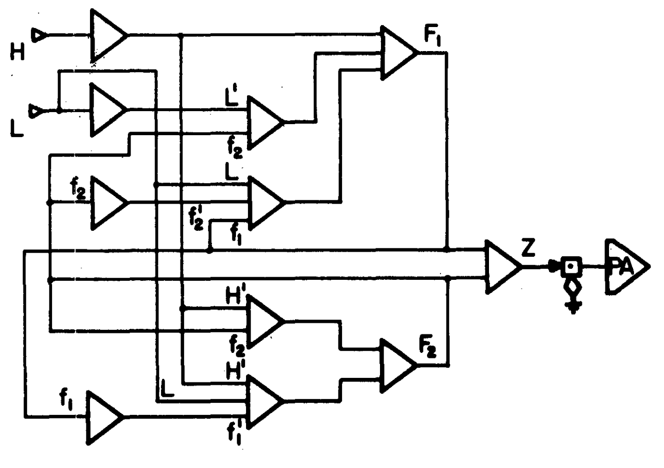


Figure 3—Pulse-height detector — Logic diagram

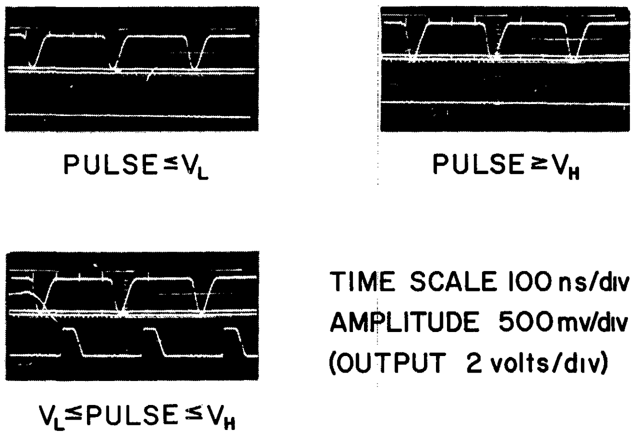


Figure 4—Pulse-height detector — Performance

A logic diagram realizing these excitation and output functions using NAND elements is shown in Figure 3. The Z function feeds a pulse amplifier which produces standardized pulses upon a logical 1 to 0 input transition. Figure 4 indicates the performance of this pulse-height detector in response to pulses which dwell at the threshold level for only some 10 to 15 nanoseconds. Notice that pulses less than the low-threshold or greater than the high-threshold produce no output. Pulses with amplitudes between these thresholds produce standard 100 nanosecond output pulses.

These output pulses are directed to one of a pair of up-counting registers in a synchronous sequential sub-system. These registers alternately store the count for the appropriate counting interval and then dump the stored count directly into a memory location using the computer data-break facility. The entire analyzer interface, which consists of some 45 flip-flops, 50 gates, and about five other miscellaneous circuits, is patched on the DADEC system. The computer controls the counting interval and keeps track of the appropriate core locations for data storage.

The computer also controls the two threshold voltages $V_H$ and $V_L$ by directing appropriate numerical values to two channels of the digital-to-analog converter. Two additional D/A channels are employed for graphical display of the accumulated count as either a function of the threshold voltages or time. This is accomplished by simply presenting these two analog channels and a device selection channel to the X, Y and Z axis of a cathode-ray-tube with storage facilities.

For this problem, and indeed for all problems of a project nature, the software support must also be developed by the students. In this instance, there are very few calculations accompanying the process and a rather short symbolic program suffices to control the experiment, accumulate the data, present the display, and punch out information for later entry into a larger computer for analysis. In this instance, the DADEC system is functioning as an on-line data retrieval system with quick-look facilities and off-line data processing.

## Play ball[2]

An interesting set of experiments is developing in the area of physiological monitoring of athletes. Thru the cooperation of the coaches and players of a baseball team, it has been possible for us to introduce strain gages and other transducers in the player's bats, switches in the player's shoes, contact assemblies in the bases and ball-speed monitoring equipment in the playing field. Small digital sub-systems have been designed and built to time the player's run to first base after the crack of a bat, to time the pitch, and to monitor the position of the pitcher's and batter's feet. The DADEC system is used to collect and correlate these data and alsoto sample and digitally represent the bat acceleration during the swing. These processes are all moderately simple and their implementation is straightforward; they will not be further described.

In this application, the DADEC system is used for data accumulation. Information is produced on punched paper tape for later analysis on large data-processing machines. For the baseball fans, a typical set of data

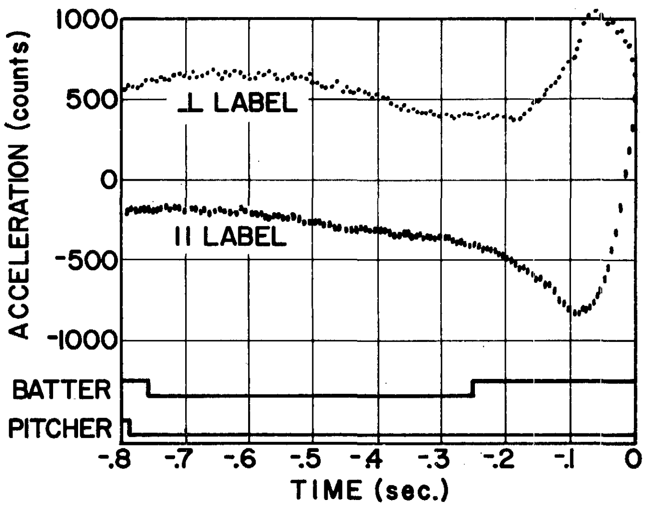


Figure 5—Typical baseball data

showing one batter's swing is presented in Figure 5. Two channels of bat acceleration are presented. One channel of acceleration is measured normal to the axis of the bat in the direction of the label, and the other is measured normal to this direction. The time at which the pitcher's foot leaves the mound (approximately the release time of the pitch) is indicated as is the shift in weight on the batter's feet. Time is measured backwards in this diagram from the instant at which the ball was hit. It might be mentioned that this is not a game situation; this trace was taken during batting practice and most pitchers would pitch faster than this in a game situation.

## Star gazing

Astronomers on campus are interested in monitoring the emitted light intensity from a star as it passes behind the moon. For a brief moment, when the star becomes eclipsed, one may observe diffraction of the light from the localized star source by the edge of the eclipsing moon. If sufficient detail regarding the diffraction pattern during an occultation is recovered from an experiment, then an apparent stellar diameter may be computed. The experimental procedure consists simply of observing the appropriate star with a telescope and focusing the total light collected from that star on a photomultiplier tube. The data recovery problem is being approached in two ways. One solution resembles the previously described pulse-height analyzer while the other resembles a portion of the baseball data recovery scheme.

The first solution method treats the output of the photomultiplier tube as a pulse source.[3] Pulses are again sent to a counter which is directed to count for a prescribed interval. At the end of this interval the contents of the counter are transferred to a shift register, the counter is cleared and again accepts pulses. In this application, the words are shifted out to an intermediate storage magnetic tape which is later read into the computer off-line. The motivation for this mode of operation arises from the requirement to develop a portable system which can be carried to the telescope sight. In this instance, the DADEC system was used as the bread-board for all preliminary design of the specialized digital sub-system. The DADEC system is again employed in the recovery of the data from the returning digital tapes.

The second solution consists simply of processing analog tape recordings of the stellar occulation.[4] The analog source is the low-pass filtered output of the same photomultiplier tube. The DADEC system controls the analog transport and accomplishes the logic for extraction of sample values from the appropriate sector of the tape.

In either experimental procedure, the end result is a number list which represents the light intensity as a function of time during the time of the occultation. For either set of data, a fast Fourier transform algorithm is applied to the sampled time functions. The relative amplitude of certain frequency components yields information from which the stellar diameter may be determined.

The final phase of data recovery for these problems is highly computational. For this reason, it is deemed desirable that the supporting software be written in FORTRAN. The students must develop facility with FORTRAN in order to establish the proper linkage for interface control and data entry within the framework of the language.

## Shocks

A final example problem to be discussed is a shock-measurement system. In this system, two pressure transducers are mounted on a moving vehicle. An air-borne shock-wave is transmitted past these two transducers. The relative time of arrival of the shock-wave at each transducer and the length of shock duration at each transducer is measured by a system which is attached to the vehicle. This portion of the system further converts this information for transmission over a telemetry link to a receiver. The typical input sequences shown in Figure 6 represent possible received signals in this system. The time $T_0$ to $T_1$ represents the shock duration time on one transducer while the time $T_2$ to $T_3$ represents the shock duration time on the other transducer. The physical reasoning is not important to our discussion, but the times of interest are the time differences $T_0$ to $T_1$ and $T_0$ to $T_2$. In some instances, for example, the second typical input sequence, $T_2$ may precede $T_0$. Notice that the two transducers modulate the signal differently so that it is always possible to identify $T_0$ as an amplitude increase of two units while $T_2$ results in an amplitude increase of one unit. Typical order of magnitude times for these events are $T_0$ to $T_1$ about 200 to 400 $\mu$s and $T_0$ to $T_2$ from about $-300$ to 800 $\mu$s. It is deduced from other engineering calculations that a resolution of one microsecond would yield sufficient information in the measurement of these time durations.

The received signal is fed to three comparators with three threshold voltages established. The comparators then yield decisions regarding the crossing of threshold level $V_A$ as a Boolean variable A, $V_B$ as variable B and $V_C$ as variable C. These inputs are further de-

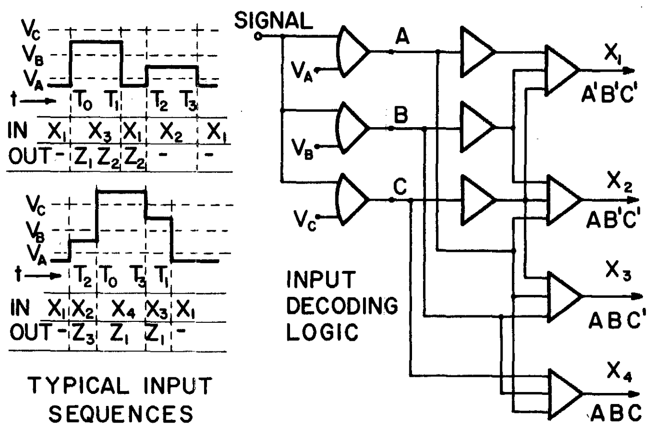


Figure 6—Shock measurement system — Front end

| $y_1 y_2$ | $X_1$ | $X_2$ | $X_3$ | $X_4$ |
|---|---|---|---|---|
| 00 | $00^{-}$ | $10^{3}$ | $01^{1,2}$ | $11^{1}$ |
| 11 | $00^{-}$ | $11^{-}$ | $11^{1}$ | $11^{1}$ |
| 01 | $01^{2}$ | $11^{-}$ | $01^{1,2}$ | $11^{1}$ |
| 10 | $10^{3}$ | $10^{3}$ | $11^{1}$ | $11^{1}$ |

FLOW TABLE

$$S_1 = X_2 + X_4 \qquad R_1 = X_1 y_2$$
$$S_2 = X_3 + X_4 \qquad R_2 = X_1 y_1$$

EXCITATION FUNCTIONS

| | $y_2 \backslash y_1$ | $X_1$ 0 | $X_1$ 1 | $X_2$ 0 | $X_2$ 1 | $X_3$ 0 | $X_3$ 1 | $X_4$ 0 | $X_4$ 1 |
|---|---|---|---|---|---|---|---|---|---|
| $FF_1$ | 0 | 0 | 1 | \| | 1 | 0 | 1 | \| | 1 |
| | 1 | 0 | O | \| | 1 | 0 | 1 | \| | 1 |
| $FF_2$ | 0 | 0 | 0 | 0 | 0 | \| | \| | \| | \| |
| | 1 | 1 | O | 1 | 1 | 1 | 1 | 1 | 1 |

EXCITATION TABLE

$$Z_1 = X_3 + X_4$$
$$Z_2 = X_3 y_1' + X_1 y_1' y_2$$
$$Z_3 = X_2 y_2' + X_1 y_1 y_2'$$

OUTPUT FUNCTIONS

Figure 7—Shock measurement system — Description

coded to produce the Boolean variables $X_1$, $X_2$, $X_3$ and $X_4$ which indicate respectively the number of thresholds which have been crossed. These signals and their logical decoding are all shown in Figure 6. The information of interest could be recovered if these X variables are fed to a sub-system which produces one megahertz output pulses on three lines called $Z_1$, $Z_2$, and $Z_3$. The $Z_1$ output should then drive an up-counter which records $T_0$ to $T_1$ time differences. The $Z_2$ output should drive the up-count line while $Z_3$ drives the down-count line of an up-down-counter which records $T_1$ to $T_2$ time differences. This will yield the appropriate time differences in two's complement binary arithmetic which is compatible with the computer.

A natural solution of this problem is hence suggested as a clocked sequential system. A flow table for such a system is shown in Figure 7. (Note that this flow table must be interpreted differently from the previous flow table and is described in references such as Marcus.[8]) The clock is not shown in the flow table, since its operation is understood. A state assignment is executed and the excitation table, also shown in Figure 7, is derived from this flow table. This generalized excitation table is of the type described in particular by Marcus.[8] From these tables the excitation and output function, shown in Figure 7 may be derived. A possible logic realization of these functions is shown in Figure 8.

This sub-system does not complete the shock measuring system. The outputs $Z_1$, $Z_2$, and $Z_3$ are fed to two counters or registers which, upon completion of an experiment, store the register contents in specified core locations by calling the computer data-break facility. The total experiment consists of observing

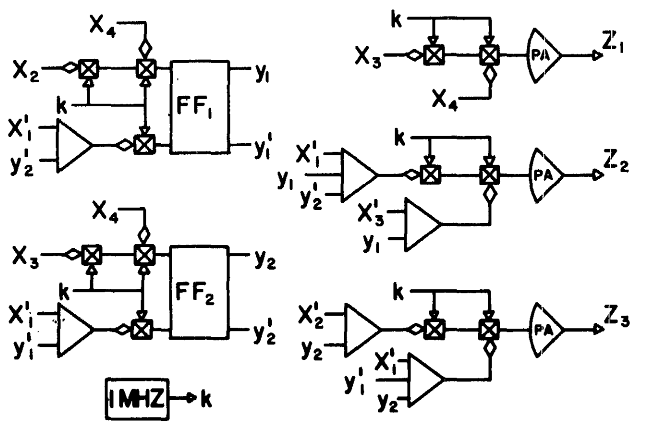


Figure 8—Shock measurement system — Logic diagram

several hundred of such shock waves which are generated in bursts at a possible rate of some 6,000 shocks per minute.

The support programming for this system was also executed by the students. In this instance, considerable calculation must be applied to the data. It was felt that the FORTRAN language was an efficient vehicle for such calculations. The FORTRAN program must communicate with the interface and such programming problems must be solved by the students.

*What's under way*

A large number of problems have been suggested for solution on this DADEC system. A listing of problems which have been accepted and are in various stages of progress is given below. It should be noted that these are undergraduate project problems and as such need not necessarily be new or spectacular in their implications. The sole requirement is that the problems have engineering application and will allow the student to follow a reasonable design procedure to achieve his goal. The problem areas under study include signal analysis using exponential basis functions, Lesbegue sampling, speech analysis and generation, automatic x-ray data processing, on-line correlation analysis, physo-acoustic reverberation studies, graphic displays, and control of psychological experiments.

### *Spin-off projects*

Several projects have developed which are not directly related to the DADEC system but are inspired by it or find use and application in design with the system. One example is a Boolean string manipulation program which accepts long strings of Boolean expressions combined with a variety of operators (EXCLUSIVE-OR, AND, OR, NOT, STROKES, etc.).[5] The string manipulation program operates on this set of characters and yields a sum-of-products type expression for the Boolean function. Boolean simplification algorithms have also been developed. A family of programs that permit a high degree of operator-machine interaction have been developed for the manipulation of flow tables.[6] These programs are useful in flow table manipulations such as the elimination of superfluous states, or accomplishing appropriate mergers and they are helpful in solving the state assignment problem.

## CONCLUSIONS

The system has been in use for about thirteen school months. Our classes are generally small; we graduate about thirty electrical engineers per year. The list of problems presented is perhaps a measure of the enthusiasm with which students have accepted this problem area and DADEC system. The anticipated problem of problem suggestion is itself no longer a problem. We are now in the enviable position of being able to be discriminating in the suggestions which we allow to go to completion. The students are beginning to vie for time on the system and in order to qualify for this time they must present an acceptable technical proposal outlining their application.

The present status of this DADEC system then is one in which a number of experiments have been developed in support of a variety of course efforts. A tremendous possibility exists for future developments of this sort. That is, the system configuration is sufficiently versatile so that only lack of the students imagination precludes his open-minded approach to a problem. It thus seems that this modest investment has sparked considerable interest and motivated the students to pursue the detail necessary to solve the problems of our new environment.

## REFERENCES

*The first six references are to student reports which are available from the Morris Library of the University of Delaware.*

1 J F BENNETT
*On-line processing of nanosecond pulses*
Dept of Electrical Engineering Univ of Delaware 1968

2 D L CLARK
*Analog and digital data recovery from magnetic tape*
Dept of Electrical Engineering Univ of Delaware 1968

3 J A BRCICH
*A stellar occultation digital data sub-system*
Dept of Electrical Engineering Univ of Delaware 1969

4 L T QUICK
*Digital processing of analog stellar occultation data*
Dept of Electrical Engineering Univ of Delaware 1969

5 L H NICHOLS
*Computer manipulation of boolean character strings,*
Department of Electrical Engineering Univ of Delaware 1968

6 G D EARLE
*Automatic flow table manipulation*
Dept of Electrical Engineering Univ of Delaware 1969

7 G A MALEY J EARLE
*The logic design of transistor digital computers*
Prentice-Hall Inc Englewood Cliffs N J 1963

8 M P MARCUS
*Switching circuits for engineers*
Prentice-Hall Inc Englewood Cliffs N J 1962

# Evaluation of an interactive display system for teaching numerical analysis

*by* P. OLIVER and F. P. BROOKS, JR.

*University of North Carolina*
Chapel Hill, North Carolina

## INTRODUCTION

The purpose of this study was to develop, use, and evaluate an interactive display system for teaching selected topics in elementary numerical analysis. We were interested in giving students a thorough intuitive understanding of the pertinent mathematical functions and in *measuring* the learning effects of an on-line graphical capability.

This system was developed in the spirit of the Culler-Fried on-line system.[1] It is similar to it in its emphasis on the combination of an interactive and a display capability, and its mathematical orientation; it differs from it in that it is designed primarily as a teaching tool rather than for problem solving.

The system developed enables the instructor or student to enter a variety of mathematical equations into the computer in a FORTRAN-like format and obtain graphical displays of these functions. In addition, the user can illustrate a number of elementary numerical methods, such as Newton's method for locating roots of equations, the Euler-Heun method for solving ordinary differential equations, and the use of interpolating polynomials. The hardware consists of a display unit with lightpen and function keyboard and a background computer. The software consists of a monitor; programs which interpret requests from the display user; and programs which produce displays.

A quantitative evaluation of the feasibility and usefulness of computer graphic techniques in teaching elementary numerical analysis raises the following questions:

1. Does the system developed perform a useful function?
2. Does it perform this function better than currently available visual facilities, e.g., slides or film? Does it help the instructor to prepare more informative and interesting lectures? Does it give the instructor more flexibility in the classroom? Does it encourage the students to take a more active interest? Does it improve student retention?
3. Can it be integrated into the teaching process so as to avoid being a distracting curiosity?
4. What does it cost to teach with such a system, and how can it be economically feasible?
5. What sort of computer system (software and hardware) is required?
6. How much manpower, time, and money is required to develop such a system?

### *Procedures*

A brief non-credit course in elementary numerical analysis was offered by the Department of Computer and Information Science in the summer of 1968. The course was held twice. One group was taught with the aid of the on-line graphic system; the other was taught conventionally. The class met for thirteen periods, two hours nightly. Prerequisites for this course were elementary calculus and a familiarity with ordinary differential equations.

The topics selected for use in the course and evaluation were

1. Polynomial approximation and interpolation.
2. Iterative methods of solving for the real roots of algebraic equations.
3. Numerical solutions to ordinary differential equations.

The system was used by the instructor to show examples during lectures and by the students in a laboratory session devoted to the properties of polynomials.

The system had been tested qualitatively by similar use during its development. We learned at that time that hands-on time by students was useful in removing the novelty of the display unit, allowing the students to concentrate on the material illustrated. It was also found that presenting a series of illustrations concentrating on a single topic, e.g., iterative methods to find roots of equations, was an effective way of imparting the key concepts of the material to the students.

## Example

The use for lecture illustration can be seen from an example. The topic roots of equations was introduced with two specific examples from physics—a column-buckling problem and a pipe-flow problem. Each problem required solving for the real roots of an equation.

Then there was a brief discussion of the techniques available for solving equations, and the field was narrowed to iterative methods. The properties common to all iterative methods were discussed, and the practical questions which face the problem solver, e.g., rate of convergence and computational efficiency, were presented.

The first specific method, linear functional iteration with acceleration, was introduced by presenting the necessary theorems on the existence of solutions and convergence.

This was followed by a series of illustrative examples. These consisted of polynomial and non-polynomial equations. The iterative method was applied to each and the regions and rates of convergence were discussed for each case. In applying functional iteration to the equation

$$x^3 + 2x^2 + 10x - 20 = 0,$$

for example, the several ways in which the iterative scheme could be set up (e.g.,

$$x = 20/(x^2 + 2x + 10), \text{or } x = (20 - 2x^2 - x^3)/10)$$

and the effects on convergence were illustrated by actually displaying each of the cases.

The Aitken acceleration scheme was then applied to each of the cases previously illustrated, and its effects on non-converging as well as converging sequences of iterates were explored.

Finally, a brief review of the techniques discussed and the key concepts discovered through the illustrative examples was given by the instructor.

This cycle of introduction, presentation of theory, illustrative examples, and review was followed in each of the classroom lectures.

Besides the lectures, each group was given a laboratory exercise designed to lead the student to the important properties of polynomials. The test group worked the exercises using the interactive display system. The students themselves operated the display device after receiving instructions on its use. The control group worked the exercises using the blackboard as a graphic device.

An examination was given on each of the three topics, as well as a final comprehensive examination covering these three topics. Each group was given a one-hour examination (the *pre-examination*) during the first day of class. This examination tested mathematical maturity and previous knowledge of numerical analysis.

Circumstances did not permit a random assignment of students to groups. Students attended the session of their choice.

The course was open to anyone possessing the necessary prerequisites. Each group was composed largely of advanced graduate students with backgrounds in statistics, mathematics, and physics, and no previous experience in numerical analysis. In each group there was one non-student. These two non-students had college backgrounds (mathematics and physics) similar to those of the students, plus professional backgrounds.

The test group was composed of four subjects; the control group consisted of six. Three additional subjects were available for measurements on the second topic, the roots of non-linear equations; two belonged to the first group, one to the second. These three subjects were given the same pretest as the others.

### *Design of the experiment*

The experiment performed was of nonrandomized, control-group, pretest-posttest design.

The two groups of observations were viewed as independent samples from a population composed of two normally distributed subpopulations. It was further assumed that each sample group was drawn from a distinct subpopulation, and that the subpopulation variances were the same, and equal to the population variance.

With these assumptions, the following tests were performed:[2]

1. A variance-ratio test for each of the post-examination results to determine the validity of the assumption of equal variances of the two groups.
2. A multivariate F-test to determine if the difference in performance of the two groups, taking the results of all four post-examinations into consideration, was due to chance or to the difference in treatments. The mean score of each group on the pre-examination was taken as the covariate, and the mean scores on the four post-examinations were the variables.
3. A t-test on the within-classes regression coefficient to determine if the difference in the initial ability of the two groups as measured by the pre-examination scores had a significant effect on the post-examination results.
4. A univariate F-test for each of the four post-examinations to test the null hypothesis

   $$H_0 : m_1 \leq m_2$$

   versus its alternative

   $$H_1 : m_1 > m_2$$

   where $m_1$ and $m_2$ are the mean scores of the test and control groups, respectively. The pre-examination mean for each group was used as a covariate. A significance level of .05 was chosen prior to performing the experiment.

*Instrumentation*

## Hardware

The IBM 2250 Display Unit, Model 1, was used for this experiment. This unit is attached to an IBM System/360 Model 40H (256K bytes) computer via a selector channel.

Images are generated by the 2250 on a cathode ray tube which has a display area of 12″ × 12″ in size, with 1024 by 1024 addressable points[3]. The following special features were available on the unit used for this experiment:

An 8K byte buffer used for image regeneration.
A character generator.
Absolute vector graphics, which allows the plotting of vectors by specifying only the coordinates of the end points.
An alphanumeric keyboard for entering characters into the buffer.
A function keyboard consisting of thirty-two pushbutton keys, an indicator light for each, and eight overlay code sensing switches.
A lightpen.

## Programming system

The graphic programming system used in this experiment operates under Operating System/360 (MFT, Version 16).

At Initial Program Load time a monitor module is loaded into a 44K partition reserved specifically for graphics. This monitor brings the application program residing in the system linkage library into the graphic partition and transfers control to it.

The graphic system is composed of seven load modules totaling approximately 5,500 S/360 assembly language instructions. No more than three load modules are ever in core at the same time. A dynamic overlay structure is used, so that at most 35K bytes of memory are used at any one time. The multiprogramming environment in which the system operates allows the user to operate while batch processing and other tasks take place using other core partitions.

The user has the following functions available to him:

General Functions:

Grid Display–

The user defines his coordinate system by providing upper and lower bounds for the x and y axes, and increments (from the lower bounds of each axis) at which he desires vertical and horizontal lines to be displayed.

Polynomial Display–

Polynomials may be displayed by entering their coefficients or their real roots in the appropriate data area. Figure 1 displays the polynomial $x^3 - x$, and shows the grid parameters along the margins of the display.

Point Display–

Up to fifteen points may be displayed by entering the (x,y) coordinates.

Function Displays–

Functions of one variable may be displayed

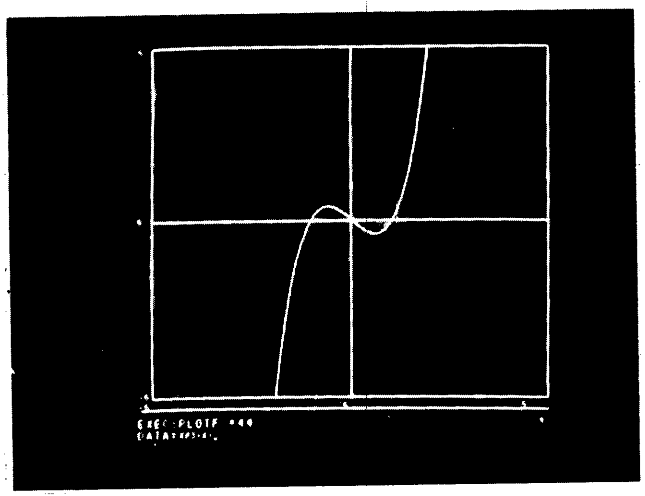


Figure 1—Display of the polynomial $x^3 - x$

by defining them in a PL/1l-ike format. Figure 2 is the display of the function tan (x) − x.

Redraw Feature–

All the polynomials in a current display, plus the most recently entered points and the most recently displayed non-polynomial function may be redrawn on a new grid.

Erase Feature–

Any single vector or set of points may be erased from the screen via use of the lightpen.

Numerical Analysis Teaching Function–

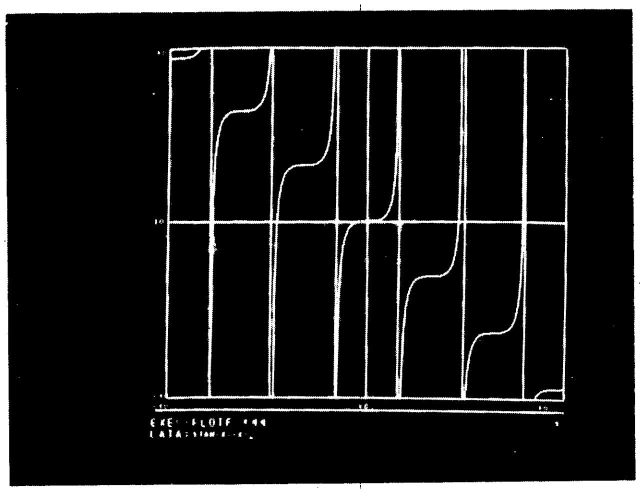


Figure 2—Display of the function tan(x) x

The following numerical analysis techniques may be illustrated:

Polynomial Interpolation
Iterative Methods for Roots of Equations

Linear Iteration
Newton's Method
Secant Method
Method of False Position

Solution of Ordinary Differential Equations

Multipoint-Methods
Predictor-corretor Methods
Runge-Kutta Method

*Using the display system*

The system was designed as a teaching tool, not a problem-solving device, although it has been used as such. Ease of use, flexibility, and hardness—i.e., the capability of continued operation in the presence of disruptions such as invalid entries by users—were prime considerations in the system's design.

Ease of use is facilitated by use of the programmed function keyboard (PFK) as the sole source of "commands" from the user—this is in contrast with using a command language via the alphanumeric keyboard, which would require the user to learn the command syntax as well as more manual effort on his part. Each command is serviced by a subroutine. This modularity of program design makes it easy to add, delete, or modify sections of code. The calling sequence is uniform for all subroutines.

The steps required to define a problem and illustrate its solution are designed to parallel those a student should perform if defining and solving the problem with pencil and paper.

The following example illustrates this. The use of a single function keyboard will be considered an "instruction," and will be designated by naming the key. (Keys are labeled on the PFK overlay.) Setting of parameters on the designated screen locations will be indicated by writing the parameter name, followed by an equal sign, followed by its value. The meta-instruction <initialize> indicates the setting of the screen dimension. In the example which follows the coordinates of the lower left-hand corner of the screen are $(-5, -5)$, those of the upper right-hand corner (5,5).

The problem is to illustrate three iterations of Newton's method to locate the real root of the equation $x^3 - x - 1 = 0$, using $x = 2$ as an initial estimate of the root. DATAPAD1 refers to a program-defined screen location used for entering parameters and functions.

Figure 3 gives the program which will generate the desired display. Figures 4—6 represent the resulting display after each iteration.

Thus, to illustrate the use of Newton's method to locate the real root of the polynomial $x^3 - x - 1$ the user performs the following steps:

1. Define the domain and range $x^3 - x - 1$ in which he is interested. This is done via the alphanumeric keyboard.
2. Use a PFK key to display the desired coordinate system.
3. Define and display the polynomial, entering its coefficient with the alphanumeric keyboard, and using a PFK key to enter this definition into main core and cause display.
4. In a similar fashion, define and store the initial estimate of the root.
5. Use a PFK key to illustrate each iteration.

These actions are those the student or the instructor would ordinarily take in solving or illustrating the problem, and are taken in the same order.

As a second example representative of the capabilities of the programming system, we illustrate the use of Euler's method for solving the differential equation

$$y' = -2xy$$

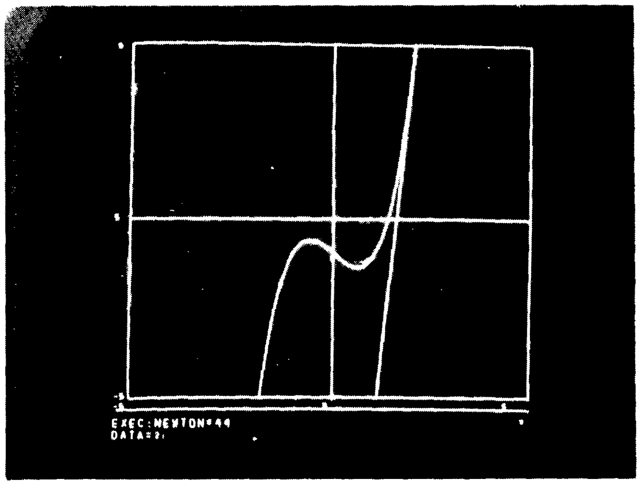
with initial condition

$$y = 1 \text{ at } x = 0$$

The domain and range are $0 \leq x \leq 3, -1.5 \leq y \leq 1.5$. A step size of .3 will be used. The large step size is chosen so as to emphasize the properties of the method.

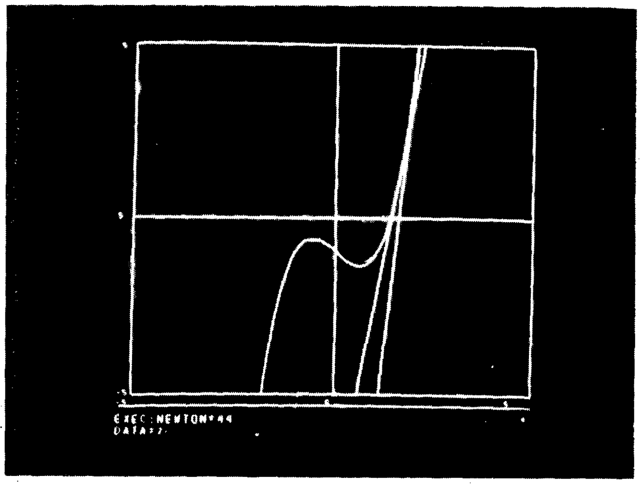
| *Instructions* | *Comments* |
|---|---|
| (initialize) | |
| DATAPAD1 = | |
| XP3 − X −1; | define function, $x^3 - x - 1$ |
| STOREF | store definition |
| PLOTF | interpret definition and plot function |
| PLACE | place cursor in DATAPAD1 area |
| DATAPAD1 = | |
| 3*XP2 − 1; | define derivative, $3x^2 - 1$ |
| STORED | store derivative definition |
| PLACE | place cursor in DATAPAD1 area |
| DATAPAD1 = 2, | define initial estimate, 2 |
| DATA | store initial estimate |
| INIT | identify stored value as initial estimate |
| NEWTON | illustrate first iteration |
| NEWTON | illustrate second iteration |
| NEWTON | illustrate third iteration |

Figure 3—Illustrative program
Illustration of Newton's method for finding the real root of $x^3 - x - 1 = 0$



```
EXEC:NEWTON*44
DATA=2,
```

Figure 4—Illustration of newton's method for finding the real root of $x^3 - x - 1 = 0$, first iteration



```
EXEC:NEWTON*44
DATA=2,
```

Figure 5—Illustration of Newton's method for finding the real root of $x^3 - x - 1 = 0$, second iteration

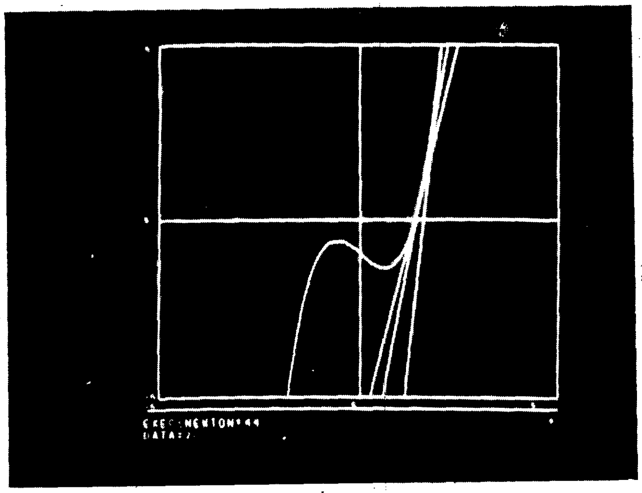


Figure 6—Illustration of Newton's method for finding the real root of $x^3 - x - 1 = 0$, third iteration

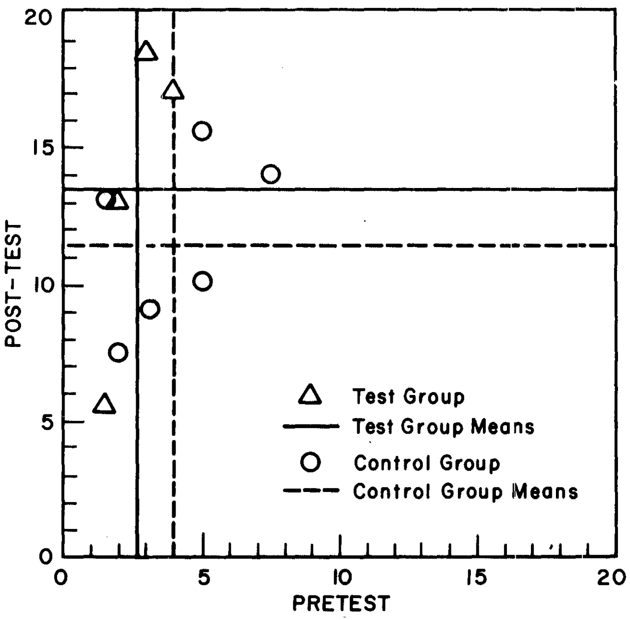


Figure 8—Scatter diagram, Interpolation and approximation

Figure 7 illustrates the approximate solution (the straight line segments) together with the true solution $y = e^{-x^2}$.

*Results of the tests*

The three subjects who participated only in the pre-examination and the roots of equations examination were not considered in performing the multivariate F-test, since the test requires that the number of subjects from a particular group be equal for each of the examinations considered. Their scores *were* used in all the other tests.

The variance-ratio test supports the hypothesis of equal variances for each of the four cases.

The result of the multivariate F-test indicates that the total differences in performance of the two groups have only a 5.8 percent probability of being due to chance. It appears likely, therefore, that the treatment differences had a significant effect on the performance

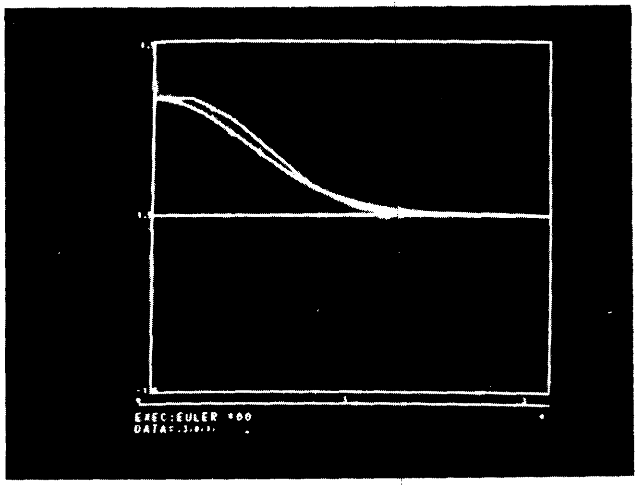


Figure 7—Illustration of Euler's method to approximate the solution of $y' = -2xy$, $y(0) = 1$ in the range $0 \leq x \leq 3$

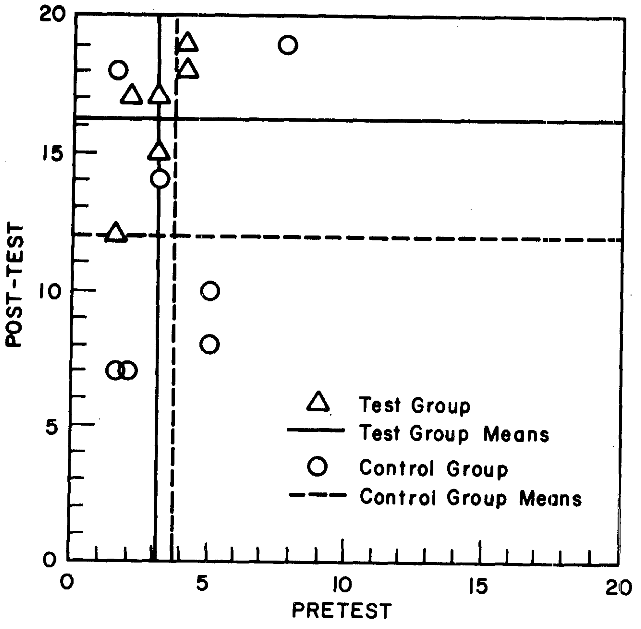


Figure 9—Scatter diagram, roots of equations

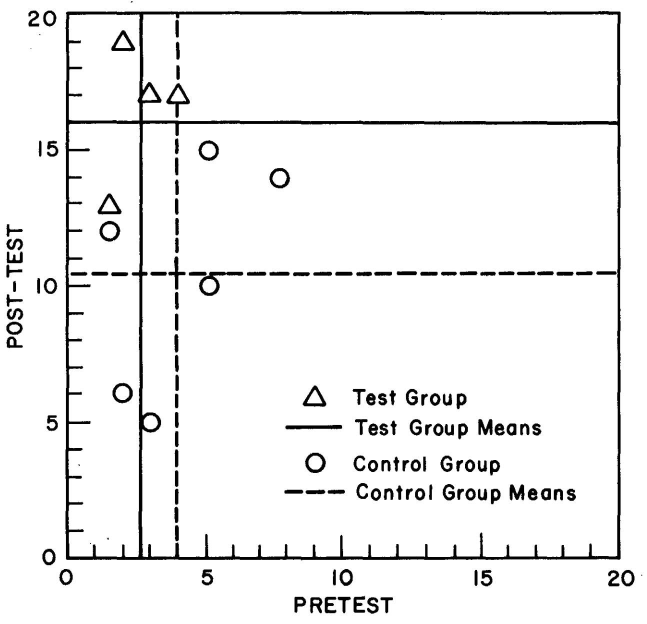


Figure 10—Scatter diagram, differential equations

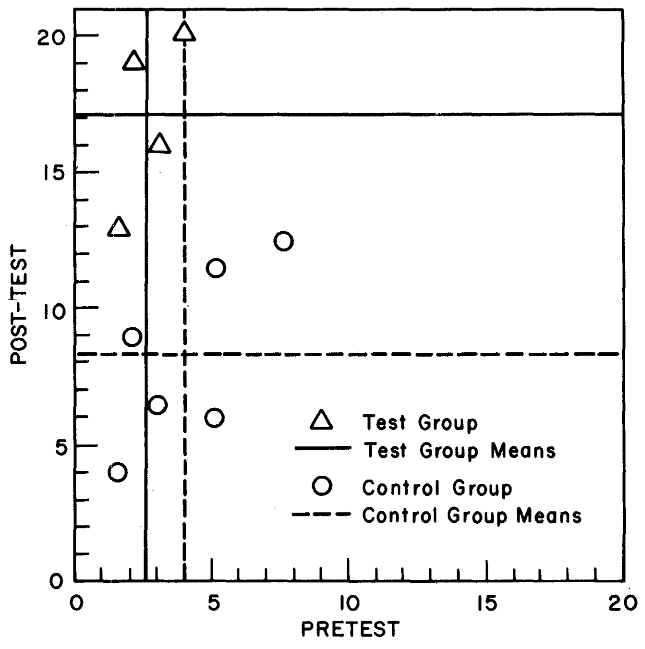


Figure 11—Scatter diagram, final examination

differences, taking all four examinations into consideration.

There was significant correlation between the pretest and posttest scores for only one of the four cases—the final examination.

Figures 8–11 give the scatter diagrams for the four examinations. The scores on each post-examination are plotted versus the pre-examination scores. These diagrams show that the test group average scores improved steadily from test to test, while the control group performance fluctuated considerably. The difference in the means for the post-examinations increased from test to test and was particularly large for the final examination. This seems to indicate that use of the graphic on-line system helped on retention, and that there was greater carry-over of learning from topic to topic on the part of the test group. The scatter diagrams also indicate greater correlation between pre- and post-examination scores for the test group.

The univariate F-tests for each of the post-examinations show that the use of the graphic system made a significant difference for the roots of equations, differential equations, and final examinations.

The data does not indicate a significant difference in performance on the approximation and interpolation examination. One may conclude that there was no difference, or else that there is insufficient data to warrant a definite conclusion. The small sample size makes the test performed very weak. Reference to power curves shows there would be a probability of .6 of error if the hypothesis was accepted that the graphic system made *no* difference.[2] A definite conclusion cannot be reached from these data on the effects of the system for the topic of approximation methods.

### *Validity of the results*

The data support the assumptions of normal distributions and equal group variances. The possible effects of previous knowledge or experience in numerical analysis were controlled by the use of a pre-examination. Even so, these effects were small. The t-tests performed on the within-classes regression coefficients indicate that the adjustment made for pre-test scores did not affect any of the raw scores except those of the final examination.

The intelligence of the subjects is the major uncontrolled variable in this experiment. It was not possible to adjust for intelligence, because scores on a common measure of intelligence were not available. If the members of the test group were much brighter than those of the control group, the experimental data could be explained thusly. Such a difference is doubtful in view of the similar backgrounds and educational levels of the two groups, and in view of the pretest scores.

Would these results apply to other groups? We cannot tell for certain until the experiment has been repeated for groups of different backgrounds, scholastic levels, and motivation. There is no *a priori* reason to doubt that it can be extended.

In summary, the following conclusions can be made regarding the quantitative results of the experiment:

1. There is evidence to support the thesis that the graphic on-line system provides a useful and efficient aid in teaching numerical methods in roots of equations and differential equations. This effect is sufficient to be demonstrated even though weak tests were used.
2. The graphic on-line capability has a positive effect on retention.
3. Further experimentation with an improved system and a larger sample must be made in order to reach conclusive results for the topic of approximation.

*Qualitative observations*

Besides the numerical data, a number of observations can be made regarding the use of the graphic system as a result of the course conducted.

1. Preparation time on the part of the instructor averaged about four hours per class hour—considerably longer than is generally required.
2. Up to twenty-five percent more time is required to present an equivalent amount of material using the graphic system than when not using it. This time is used in setting up illustrative displays.

   This set-up time is distrarting to the student. Intermittent use of the graphic device during a class session is especially distracting. A good procedure is to introduce the material briefly, present the necessary theorems; give a series of examples illustrating the methods an algorithms; terminate the session with a brief summary of the material.
4. The amount of information displayed is important—each display should illustrate at single principle rather than several.
5. The ability to regenerate an entire display on a changed grid size proved very useful. The instructor can illustrate a particular problem in the large, and then enlarge a particular part to fill the entire screen.
6. A system will fail at times. The instructor must be ready to continue the illustration in progress at the blackboard. He must be thoroughly familiar with the problems he is presenting.
7. Whenever possible the instructor should encourage the students to discover the point of a display.
8. Hands-on time on the part of the students is very useful. One problem of the final examination consisted of determining the parameters a, b, and c in the polynomial form $a(x + b)^2 + c$ so that the resulting polynomial would pass through three given points.

   The test group handled this with ease, and each individual was able to find the correct values and explain the steps taken to arrive at them. Most of the control group subjects were not successful, and those that were were not systematic in their approach. The purpose of this exercise was *not* simply to find the coefficients. Rather, it was to illustrate the effects of varying the three parameters on the behavior of the polynomial.
9. Class participation was much greater in the test group. The students in this group were eager to pursue topics which were not directly covered in the lectures. During the lecture on iterative methods for finding roots of equations the students in the test group discovered the effects of applying acceleration to diverging sequences of iterates, and did so by their own initiative. The test group also worked the examination questions much faster than the control group, usually starting by drawing a picture.

*Findings and conclusions*

Experience to date gives tentative answers to the questions initially posed:

1. The results indicate that the interactive display system is a valuable and powerful aid in teaching selected topics in numerical analysis.
2. The system performs this function better than visual facilities generally used. The graphic and the interactive capabilities enable the instructor to develop a large number of significant examples to illustrate his classroom lectures and to make them more interesting. The interactive capability provides a flexibility not available through slides or filmstrips. Complete response to student questions stimulates student inquisitiveness. Student retention is improved, and there is a greater carry-over of learning from topic to topic.
3. The system can be effectively integrated into the teaching process, but delay time—the time necessary to generate new displays—and reliability are problems which require an unusual level of instructor preparation.

4. The cost of teaching with such a system is not high except for the cost of the display unit. Running the system requires very little processing time. Preparing class problems requires about five minutes of Model 40 CPU time per display hour. Classroom presentation averaged about two minutes of CPU time per display hour. The display unit is costly, but this application could use a simpler and cheaper display device. Both cost and reliability can be improved by using this system to prepare slides for classroom use, but extemporaneity and flexibility will be sacrificed.
5. In determining the hardware and software capability required for such an interactive display system, a number of items must be considered. A 12″ × 12″ screen size is about average for display units with vector capability. A smaller screen size could be tolerated for individual use, but not for classroom use. The alphanumeric keyboard is essential for entering data into the system, but the function keyboard could be eliminated. One could use the standard alternative of a menu of lightpen buttons displayed on the screen. One could *not* readily substitute the alphanumeric keyboard for function buttons without seriously impairing ease of use. The 8K buffer used in this experiment could be reduced to 4K without impairing system efficiency.

   A graphic programming support such as the IBM Basic Programming Services is useful but not vital. The applications facilities required would depend on the use to be made of the system. Those used in this investigation were minimal though adequate for teaching the selected topics in numerical analysis.
6. Development of the system described here required about 1200 man-hours, with one individual devoted to this task over a one-year period. Development also required about 163 hours of S/360 Model 40 time.

The results of this experiment indicate that use of an interactive display system can significantly increase the active role of the learner and improve student insight and understanding of elementary topics in numerical analysis.

This is a pilot study. It demonstrates the usefulness of such a system only for one group of students with one particular subject-matter. To generalize, one would have to replicate this experiment with other groups of students.

The study is, however, as useful for what it suggests as for what it proves. It suggests specific techniques for using such a system. It suggests that we measure the separate effect of student hands-on time. A controlled experiment should be run in which students use the graphic system to work a given set of problems, studying a set of notes presenting the necessary background material. This treatment would not involve an instructor except as a monitor.

Finally, the study suggests the desirable characteristics of follow-on systems and ways of making them more economical.

## REFERENCES

1 B D FRIED
*Solving mathematical problems*
McGraw-Hil Book Co Inc N Y 1967 In On-line Computing edited by W. J Karplus

2 B J WINER
*Statistical principles in experimental design*
Mc Graw-Hill Book Co Inc N Y 1962

3 *IBM System/360 component description, IBM 2250 display unit model 1*
IBM Corp Form A27-2701 1969

4 J C R LICKLIDER W E CLARK
*On-line man-computer communication*
Proc SJCC Vol 21 1962 113-128

# Computer based instruction in computer programming—A symbol manipulation-list processing approach

by P. LORTON, JR. and J. SLIMICK

*Institute for Mathematical Studies in the Social Sciences*
Stanford, California

## INTRODUCTION

Since February, 1969, a computer based course in computer programming has been running at an "inner city" high school in San Francisco, California. Each day ninety high school juniors and seniors in classes of fifteen interact with a course designed to teach the fundamentals of computer programming for business applications. For fifty minutes a day each student is on-line with a computer located thirty miles away on the Stanford University campus. The purpose of this paper is to describe the rationale and the major components of the software system used to implement the project.

Lesson material and programming problems for the students are presented on teletypewriters linked via telephone lines to the Computer Based Laboratory of the Institute of Mathematical Studies in the Social Sciences on the Stanford University campus. In this laboratory are several computers which form a unique system for presenting instructional material.

The main computer in the system for this project is a Digital Equipment Corporation model PDP-1D. The PDP-1D is a single address, 18 bit binary machine. The machine has 32,768 words of core memory of which 20,480 words are used by the time-sharing operating system. User programs are permitted up to 12,288 words of core. The time-sharing system allows up to 26 users to run concurrently on the computer. This is made possible by the addition to the PDP-1D of a very high speed drum with 26 tracks, each capable of holding 4096 words. The time-sharing system swaps programs in and out of core memory very rapidly using a simple priority scheme based on "time-slicing." Because of the necessity for user micro time-sharing the programs in this project occupy 10 of the 26 available tracks.

The PDP-1 communicates with the students at the high school through a smaller computer (DEC PDP-8) used to buffer text output. A PDP-8I has been installed at the school to perform a similar function at the other end of the line. Collins data sets were used in place of the PDP-8I during the first year.

### *Aim and purpose of the course*

The main goal of this course is to present in very general terms the concept of a digital computer as a tool for solving business-related problems. As computers proliferate in business and industry there will be an increased demand for people who can see their jobs in terms amenable to computerized operation. Such tasks as filing and stockroom control, now available to minimally trained individuals, will soon require personnel able to see and solve problems in terms understandable to a computer.

With the goal of training for applications on these kinds of problems, the need for something other than a "formula translation" approach is evident. Using filing and stock control as sample problem areas, an approach which stresses symbol-manipulation and list-processing suggests itself. Inventories can easily be viewed as ordered pairs (a symbol-manipulation concept) of item names and counts. Retrieving information from a file can be thought of as a "tree search" (a list-processing concept).

The advantages of teaching a symbol manipulation-list processing (abbreviated: SMLP) language are best shown in an analysis of the properties of SMLP languages.

A. SMLP languages operate primarily on symbols and sets of symbols and, secondarily, on quantities. This implies that problems as conceptually complex as text scanning become more manageable. Once text scanning becomes manageable, then many applications such as natural language-based information retrieval or dialogue systems for management information collapse into programmable problems. The power of an approach which emphasizes symbol manipulation is that conceptually difficult problems often become readily programmable.

B. The list structure in SMLP languages provides an absolutely general form of data and program storage. A programmer, given a universal data storage facility, can give some attention to optimization of the structure of his data. The optimization of data structure cannot be over emphasized since information retrieval (among other applications) is not economically possible without structuring the data so that the computer answers efficiently the most frequently asked questions.

C. SMLP languages teach the use of pointers and indices. While properly part of (B), the simplest definition of a pointer is that it is a quantity that specifies the location or existence of some other quantity; an index can be defined as a quantity specifying some base location. The concepts of pointer and index are useful in teaching the manipulation of data by using references rather than moving blocks of data from one place to another. An immediate example of an application of pointers is data sorting.

D. SMLP languages allow simple implementation of push-down stacks. While not of great intrinsic value, push-down stacks simplify the calling and structure of subroutines, particularly recursive ones.

E. SMLP languages simplify the treatment of name scope problems in a hierarchical store. A fundamental concept of symbolic programming is that a quantity can have a name; furthermore, it may be desirable to limit the area of the program in which a given name refers to a particular quantity. Thus, it is desirable to have a method of associating a given name to the relevant quantity on the basis of "area"; this association is referred to as "name-scope."

In general, language possessing properties A-E provide exceptionally general approaches to programming digital computers. It can also be pointed out that the COmmon Business Oriented Language (COBOL) resembles this kind of language more than it resembles a "formula translation" language. The general concepts available through an SMLP language would, it is believed, be of considerable help to the students in their future efforts to build an understanding COBOL and related languages.

## Basic concepts

Good computer programming, under the philosophy advanced here, depends on the understanding of certain concepts not particularly oriented toward any one machine or language. The basic concepts which seem necessary for understanding the kind of applications programming taught in this project seem to divide into concepts which are related to making a stored program machine work for the user and concepts which are related to what is felt to be the basic task of business applications programming: symbol manipulation-list processing. It is these concepts which form the basic content for this course.

The first nine general concepts in the following list are of the first type. The tasks described are all associated with the how and why of making stored program machines do the work required of them.

I. "Machine" related concepts:

A. Stored Program. Refers to the ability to have a set of imperative actions implying some overall task stored in a machine which can execute it in some sequential fashion.

B. Stored Data. Refers to the ability of a machine to store quantities like "stored program" actions but not encompassing an overall meaning.

C. Variable. Refers to the ability to name some part of the stored program and refer to the properties or value of this part through reference to its name.

D. Operations. Refers to the capabilities contained in the Central Processing Unit. Two main classes of operations are felt important: Arithmetic and Non-Arithmetic.

E. Addressing. Refers to the capability of pointing to various parts of the stored program as well as the ability to form data into clusters or arrays in some useful way. Three sub-concepts are felt noteworthy: Indexing, Base addressing, and Indirect addressing.
F. Branching. Refers to the ability of a stored program to reorder the sequence of events it performs in completing a task.
G. Loops. Refers to the ability to re-execute a subsequence of the stored program to complete a repetitive task.
H. Blocks/Sub-Programs/Procedures. Refers with minor differences in emphasis to sub-groupings of the stored task which form semi-self contained programs often capable of being introduced into the main event sequence by being "called."
I. Input-Output. Refers to the machine's methods for listening and talking to the user.

The following concepts are more directly related to the symbol manipulation-list processing approach to the problem space than they are to the problem of making a machine work. This does not mean that the concepts listed above are unrelated to issues associated with the nature of the problem space. Neither does it mean that a ymbol manipulation-list processing language is unsuited to presenting them.

II. "Language" related concepts:

A. Data Handling. Refers to the method of viewing and manipulating the data a program is to handle.
B. Recursion. Refers to a "self calling" ability of sub-blocks of the program in an SMLP type language.
C. Arrays and Strings. Refers to a more general and efficient way of clustering stored data so that its manipulation becomes a simpler task.
D. Data Structures. Refers to named functions which use indexing and pointers to locate elements in the stored data. Examples might be "trees," "lists", "graphs", etc.

## Languages selected for the project

Given the conclusions on the advantages of teaching a "symbol manipulation-list processing" language and the fact that some machine level concepts might usefully be introduced into the course, a language appropriate to each conclusion was selected: a simple assembly language and a fundamental SMLP language. Each of these languages is briefly described below.

*Major components of the project*

The implementation of the conclusions reached in the preceding discussion involved developing three separate programs which, when loaded into the PDP-1-D, operate as the software system for this project. The three programs include a "driver" (SLAKER) to supervise the interaction of the student with the curriculum material and the language processors, an interactive assembly language processor (SIMPER), and an interpretive SMLP language processor (SLOGO). Each of these parts of the software package is described below. Appendix A contains a sample lesson illustrating many of the components described below.

## Major component: SLAKER

### Introduction

SLAKER [Slimick-Lorton All Knowing Educator Routine] is designed to provide the interface between the student at a teletypewriter and the curriculum material of the project. The over-riding concern in the development of this driver was to provide as much freedom and flexibility for each user as is consistent with service at reasonable intervals.

If a student's program would cause a real machine to enter an infinite loop or write over his data, then this would happen to him in the instructional setting. Certain obvious restrictions have been placed on this goal. A student's work is not free to "clobber" other users (although this might well happen on a "real" machine). A student can wipe out his own effort and experience the pain of having to recover from the error.

### Functions

The balance of the description of SLAKER is devoted to the major functions it is designed to perform.

1. Text Emission

One of the major tasks SLAKER has is the presentation of problems to the student at his teletypewriter. Several of the disc files attached to the driving program contain the curriculum material which is organized into four sequences of lessons and problems through which the student is to proceed. In addition to the lesson-text, the problem code contains certain values which indicate various subsections of the problem such as the "correct answer" or the "hint," as well as the problem type to the driver.

The four strands into which lessons and problems are grouped for this project are: Lesson, Homework, Extra Credit, and Test. For problems in the Lesson strand, SLAKER is charged with waiting until the student enters the correct answer before going on to the next problem. With the other three strands, SLAKER presents the next problem as soon as any answer is entered. In every case the student is informed of the correctness of his answer.

2. Response Evaluation

After emitting the text for a problem to the user, SLAKER monitors his output, collecting it as an answer. When the user enters an "evaluate my work" request, SLAKER checks his answer according to the type of problem the student was given.

A. Multiple Choice

Under this format the answer is first compressed so that all duplicate characters are eliminated. Then the answer is searched for matches with the characters recorded as the correct answer. Up to twenty characters are collected from the student as possible answers for problems stated in this format. Only alphabetic characters are collected so that spaces, punctuation marks, or numbers can be inserted in the answers without affecting the correctness of the alphabetic string.

B. Constructed Response

When the student's input is a response to this type of problem, all the characters he types, with the exception of carriage returns and line feeds, are collected. The checking routine then examines the response string looking for two kinds of characters: those that must be present and those designated as optional. The serach and match routine is of such generality that it is felt all possible correct answers will be marked correct if they are defined in the curriculum.

C. Anticipated Alternative

Although not a separate type of problem, this checking capacity is a separate skill of SLAKER. If alternative answers are expected they can be specified and checked for. If a correct response is not found, then the answer evaluation routine checks the student's effort, in the same fashion, against the strings specified as possible alternatives. If a match is found, then an appropriate comment is given and the student is told to try again, just as if he were wrong. At present this capability is available on constructed response problems and single choice-multiple choice problems.

D. Programming Problems

Evaluation of these problems is done by asking the student questions about his program after he wrote and debugged it with given data. This method of evaluation allows the student flexibility in programming a different solution than the solution the curriculum writers had in mind.

3. Communication with Language Interpreters

Since the main aim of the course is to provide rich and varied experience in programming, a main responsibility of SLAKER is readily to provide this contact. Each language differs slightly in how it wants to be told a student is using it but, basically, SLAKER's role is to make the initial contact with the language processor, pass subsequent information to it and await the user's indicated wish to return to the main program.

4. Special requests from the Student Station

The following activities can be requested from a student station. As a group they provide the student with considerable flexibility in how he proceeds through the course.

A. Restart Station. Allows a user to request a station be restarted from the sign-on point. Used to correct improper sign-on efforts by students.
B. Sign-off Station. Allows a user to terminate his lesson when he is ready. Part of the execution of this command involves storing where the student left off on his history file so that he may restart from this point on the following day.
C. Go to Choice Point. Places the user at a point where one of the following choices can be made:
    1. Return to Last Problem. Allows the student to continue working from where he last signed off in the strand he specifies.
    2. Go to Specific Lesson. Allows the student to begin working on the lesson number in the strand he indicates.
    3. Attach a Language Processor. Allows the student to call forth one of the language processors available in the course.

D. Skip Problem. In the Lesson strand, only a correct answer will advance a student on to the next problem. This feature allows a student to skip out of this loop. As the next problem is called, the correct answer to the skipped problem is printed.

E. Give Hint. Commands SLAKER to print the "hint" provided for the particular problem.

F. Erase Answer. The user has the option of erasing all of the answers he has typed or merely the last character. Erasing the last character can be repeated until the entire answer is erased if wished.

G. Communicate with Stanford Monitor. This feature allows student stations to type messages to the monitor teletypewriter at Stanford. Usually, its use is reserved for the classroom teachers who may want to correct a lesson, enter a new student, or ask a question. As part of this feature it is also possible to communicate from the monitor teletypewriter to any of the student stations.

## Major component: SIMPER

### Introduction

SIMPER [Simple Instructional Machine for the Purpose of Educational Research] represents an attempt to make available to the student at a teletypewriter a simple computer which he can program in a manner analogous to "assembly language programming" on digital computers of modest size.

This instructional package can be most easily understood when viewed as consisting of two main parts: a machine (SIMPER) and an assembler (SASS). The latter is designed to generate the machine code for SIMPER. The "machine" is a mythical digital computer which can be described in a formal way and for which programs can be written. Although the machine responds to 18 bit instructions in its "machine language," there is no direct access to the machine via 18 bit numbers. The purpose of the machine is to teach students to program so the machine is programmable only through a symbolic assembly language.

The assembler generates code for SIMPER from Assembly Language instructions typed by the student. Assemblers generate code instruction by instruction. This one generates code for SIMPER immediately after each instruction is typed in by the student. This feature enables the student to receive immediate correction for most syntax errors and, when the student avails himself of the option, each line of code can be checked immediately to assure the student that the assembler translated the student's instruction as he wished.

The current version of SIMPER is designed to time share up to 15 students concurrently. The interpreter occupies 4096 words of PDP-1D core memory while the arrays representing the simulated machines for all 15 possible users occupy an additional 4096 words of memory.

### Description of the SIMPER machine

SIMPER is a fixed-point, single address machine with a memory of variable size (currently 128 words). Operations are performed in two general purpose registers. Instructions are six digits in length: two digit operation code, one digit register specification field, and a three digit address field. At present, 16 operations can be performed.

The size of the machine's memory is variable depending on the available space. For this project the memory size is 128 decimal (200 octal) locations. This size was chosen because it allows the fifteen students to run parallel in the space available on the PDP-1D and it also means the students' daily programming effort can be "saved" on a disk scratch file of convenient length, enabling the student to continue programming efforts from session to session.

### Operation of the SIMPER machine

SIMPER runs by executing the six digit number it finds in the memory location pointed to by the program counter. The program counter is updated as part of the instruction-fetching activity. An instruction by instruction-execution of a program is printed on the Teletype. While thus being able to monitor the execution of his program, hopefully, a student is given special insight into how each instruction operates and how a sequence of instructions can be converted into meaningful work. This "printing out" of the execution sequence also slows down the speed of execution so that the work of the machine is easily followed. The student can also watch the effects of "bugs" arise and develop into problems which require attention. This feature is intended to make the debugging of machine language programs an easier task. A special flag can be set at execution time to suspend this feature. Execution speed is then improved by a factor of four.

*The assembler*

**Description**

The assembler recieves its instructions from a student through a teletypewriter keyboard. Each student interacting with the program is listened to for characters which are collected as an instruction to be assembled. Students are served by the assembler in a manner which both time shares and "oils the squeaky wheel first."

When the student is given a problem involving assembly language programming, he is told to sign on to SIMPER. He calls the choice point option and, in response to "Where to?→", types "SIMPER." The student is then in contact with the assembler. He is informed that he may now write his program and columns labeled "LOC" and "INSTRUCTION" are created. In the LOC column the assembler prints the number of the memory location into which the instruction being written will be assembled. The assembler then awaits an instruction from the student. The student types his instruction and an indicator that he is finished. The assembler immediately examines the text string and attempts to generate SIMPER executable code. If all is in order, programming advances to the next memory location. If all is not in order, the assembler generates an appropriate error message. By assembling in real-time after each instruction is entered, the assembler can give immediate feedback on syntax errors to the student.

## Major component: SLOGO

SLOGO (*S*tanford **LOGO**) is the I.M.S.S.S. implementation of LOGO, a computer language developed by Wallade Feurzig and Seymour Papert of Bolt, Beranek, and Newman expressly for teaching the principles of computer programming. SLOGO is similar to LISP 1.5 in that both are left prefix languages, both have a simple type of function definition, and both have similar sets of primitive operations. SLOGO functions, unlike LISP, have predefined numbers of arguments which, along with the left prefix notation, allow SLOGO to require minimal user punctuation.

While SLOGO is an ideal symbol manipulation and string processing language, it has substantial weakness in not providing structures that are effectively lists of lists *à la* LISP 1.5. While generality is very desirable to the programmer, the choice of LISP 1.5 as the symbol manipulation-list processing language for this project posed such severe curriculum problems that the attempt to use it was abandoned; thus, SLOGO, which has less generality, was implemented instead.

SLOGO currently time shares five concurrent users; each user has a 4096 word drum track that contains his own functions, execution stack, etc. SLOGO is a re-entrant program when executing commands from a user, but it is not re-entrant with respect to console input and the queuing apparatus. The currently available functions with short definitions attached are listed in Appendix B. In the following sections, first, the basic data types used in SLOGO are described, and immediately thereafter is a discussion of the two processing modes of SLOGO.

### Data types in SLOGO

There are three basic data types in SLOGO: word, sentence, and number. A brief explanation of each follows.

(1) A "word" consists of a string of letters, digits, or certain punctuation marks; punctuation marks that cannot be used are blank, single quote, ">", "<", "-", and possibly others that depend on which version of SLOGO is being run.

(2) A "sentence" consists of a group of words. Although one can argue that sentences could consist of one or more words, to avoid ambiguity we assume that sentences consist of two or more words.

(3) A "number" consists of a string of decimal digits plus a leading minus sign, if the number is negative. The largest number acceptable is ± 131,071.

There are three methods of referring to data: function values, pointer variables, and literals. A brief explanation of each follows.

(1) A literal is a direct reference to the indicated data. Word and sentence literals are written with the single quote (') surrounding the desired data. Literal numbers appear as the number itself, without quotes. A quoted number is assumed to be a word.
Example:

The following are word literals:

'AARDVARK'
'45'
'3A'
'MIXTEC'
'THISISAWORD'

The following are sentence literals:

'AARNOLD IS A APATHETIC AARDVARK'
'ONTOGENY RECAPITULATES PHYLOGE-

NY'
'1 2 3 4 5'
'THIS IS A SENTENCE'

The following are number literals:

1
1776
−10
131071

(2) Function values. Most of SLOGO's built-in functions and all of the defined functions return a value. This value may be subsequently referenced by other functions, and the type of this function may be any of the three basic types.

(3) Pointer variables are in reality name pairs, where one part of the pair is the name and the other part is the value. Names must have type values of either word or sentence but never number. The value type can be word, sentence, or number. Names are written inside closed symbols, which can be either "<" and ">" or "—" for left and right sides.

Example:

<ANTEATER>
<NURNDY IS A GAME>
—POINTER—
<A>

The peculiar literal '' is accepted by the read-in routines, can be generated internally, and is always printed by SLOGO as "NIL".

To illustrate the difference between literals and pointer variables, assume there is a name pair whose name is "HEROINE" and whose value is the sentence "OUR GAL SUNDAE."

The value, then, of <HEROINE>

is OUR GAL SUNDAE.

The value of 'HEROINE'

is HEROINE.

### SLOGO processing modes

SLOGO operates in two modes, command and definition. There is a special character printed at the extreme left-hand end of the type line to indicate which mode SLOGO is in.

"Command" mode is indicated by a ">" ("greater than") sign, and is the normal mode of operation. In command mode, as soon as a line of functions and arguments is typed in, terminating with a "." (period), the line is converted to a Polish string of interpretive code and then interpreted by the SLOGO interpreter. Upon detection of an error or the successful execution of the Polish string of code, whatever output produced is printed (if PRINT is used) and SLOGO returns to a listen state while the next line is being typed in.

"Definition" mode is indicated by a "→" ("right arrow") sign, and is the exceptional mode of operation. It is entered from command mode when an input line has been terminated with a period and begun with a "TO". At that point definition mode is entered and cannot be left until the command "END" is entered. There is no attempt at function execution while in definition mode. The only use of definition mode is to define a SLOGO function by entering successive lines of functions and arguments. During definition mode, checking is done on the function names, validity of arguments, etc., but no functions are executed.

## SUMMARY

The purpose of this paper has been to describe the software and corresponding rationale for a project designed to teach high school students how to use computers. The main thought behind the project is that, especially for business applications, an approach which stressed symbol manipulation and list processing skills would very likely prove of long-term use to the students.

To implement this course, a three-part software package has been developed which provides guided interaction for each student with important programming concents. The software package includes a "driver" to shepherd the student through the course material, an assembly language interpreter to provide him with an understanding of basic machine operation and a symbol manipulation-list processing language interpreter to provide him with experience in solving problems in a suitable higher level language.

It is worth noting that all of these programs are written in a subset of ALGOL-60. A course dedicated to the teaching of higher level computer languages could show the utility of such languages in no better way than to have its software packages written in such a language. One of the very useful demonstrations this project has made has been to show that complete, useful and efficient computer-based instruction systems can be written in a higher level language.

Preliminary and informal results from the students in the course are quite encouraging and tend to support the basic philosophy of this approach. There is every reason to believe that the future statistical analysis of the effects of this course will confirm these initial observations.

## APPENDIX A

*Sample lessons*

(The following are short examples from the actual curriculum; they have been retyped. Comments within brackets are parenthetical comments added to indicate various features.)

```
3 JULY 1969
SLAKER (VERSION OF 28 MAY 69)                                  [sign-on]
PLEASE TYPE YOUR NUMBER···→ 11
(CTRL G TO BEGIN-CTRL T TO RESTART) →                          [start at Lesson 68]

WHERE TO? → L68
.........................
LESSON 68: USING TESTS                                         [a SLOGO lesson]
.........................
WE CAN USE 'FIRST,' 'BF' AND SO ON WITH 'CALL'
IF YOU TYPE THIS:
    CALL FIRST OF BF OF 'BEARS HIBERNATE IN WINTER' 'X'
    IF WORD? < X > THEN P < X >.
THEN SLOGO FINDS THAT < X > IS 'H,' WHICH IS A WORD,
SO SLOGO REPLIES:
    H
FOR PROBLEMS 1-6, TYPE WHAT SLOGO REPLIES
TYPE 'N' IF NOTHING IS PRINTED.
  1. CALL FIRST OF 'BLUE SKIES' 'W'
     IF WORD? < W > THEN P < W >.
     → BLUE···CORRECT                                          [constructed response]
  2. CALL DIFF OF 9 AND 6 'X.'
     IF NUMBER? < X > THEN P TIMES OF 4 AND < X >.
DOES SLOGO THINK 3 IS A NUMBER?                                [a hint]
→ 12···CORRECT                                                 [another constructed response]
NOW SIGN ON TO SLOGO AND DO PROBLEMS 7-10.
AFTER ALL 4 PROBLEMS ARE DONE, TYPE CONTROL A.
  7. TEST TO SEE IF 'PLACE KICK' IS A SENTENCE.
     IF IT IS, PRINT 'IS SEN.'
  8. TEST TO SEE IF 7 IS A WORD. IF IT IS, PRINT 'IS WORD.'
  9. TEST TO SEE IF '1 4 8' IS A NUMBER. IF IT IS, PRINT THE NUMBER.
 10. TEST TO SEE IF 'P' IS A WORD. IF IT IS, PRINT BF OF THE WORD.
     →
WHERE TO? → SLOGO···OK                                         [sign-on to SLOGO]
SLOGO···THE ORIGINAL CONJURING CAT                             [hello from SLOGO]
> IF WORD? 'P' THEN P BF 'P.'                                  ["P" is "PRINT." ("BF" is "BUT-
= NIL                                                            FIRST"); this is solution to 10 above]
> TO REVERSE < A >.                                            [Sample of SLOGO programming]
    → IS < A > '  .'
→ IF YES RETURN '  .'
→ RETURN WORD LAST < A > AND REVERSE OF BUTLAST < A >.
→ END.
> P REVERSE '1234567890.'
= 0987654321
>···OK                                                         [return to SLAKER]
```

```
OUTPUT SHOULD BE:
  7. IS SEN                                         [correct answers to 7–10; control
  8. NO OUTPUT                                      S ("SKIP") takes one on to 11]
  9. NO OUTPUT
 10. NIL
................................
LESSON 11: PROBLEM SOLVING                          [a SIMPER lesson]
................................
WRITE A SIMPER PROGRAM TO SOLVE EACH OF THESE PROBLEMS FOR YOU
  1. MARY BOUGHT 3 POUNDS OF CANDY AT 29 CENTS PER POUND.
     WHAT WAS HER BILL?
     →
WHERE TO? → SIMPER···OK                             [go to SIMPER]
SIMPER (VERSION OF 6 JUN 69)                        [hello from SIMPER]
BEGIN PROGRAMMING
LOC INSTRUCTION
000 → BEGIN                                         [possible student solution to this problem
001 → GET X
002 → GET Y
003 → LOAD X
004 → MUL Y
005 → STOR X
006 → PUT X
007 → END
008 →
EXECUTE···STARTING LOC → 0 AND ENDING LOC → 7
PROGRAM EXECUTED ON 3 JULY 1969
    P C  INSTR   REG A   REG B
    000  BEGN        0   32768                      [execution of solution]
INPUT → 3
INPUT → 29
    000    LOAD      3   32768
    004    MUL      87   32768
    005    STOR     87   32768
OUTPUT = 87
    007    END      87   32768
···END OF EXECUTION, CONTINUE
    008 → ···OK                                     [back to SLAKER]
HER BILL WAS 87 CENTS. IF YOUR PROGRAM SAID         [answer]
OUTPUT = 87, SKIP ON.
    →                                               [skip on]
  2. A RECTANGLE IS 8 INCHES LONG AND 4 INCHES WIDE.
     FIND ITS AREA.
     → TO FIND THE AREA OF A RECTANGLE, MULTIPLY THE LENGTH         [hint]
     TIMES THE WIDTH.
  →
WHERE TO? → SIMPER···OK                             [sign-on to SIMPER]
```

## APPENDIX B

*Concise guide to SLOGO*

(Optional words are italic).

| | |
|---|---|
| WORDS *OF* X *AND* Y | produces a word which is X concatenated with Y. |
| SENTENCE *OF* X *AND* Y | produces a sentence of Y appended to X. |
| FIRST *OF* X | if X is a word, result is the first letter; if X is a sentence, result is the first word. |
| BUTFIRST *OF* X | if X is a word, result is all but the first letter; if X is a sentence, result is all but the first word. |
| LAST *OF* X | if X is a word, result is the last character; if X is a sentence, result is the last word. |
| BUTLAST *OF* X | if X is a word, result is all but the last character; if X is a sentence, result is all but the last word. |
| SUM *OF* X *AND* Y | $X + Y$ |
| DIFFERENCE *OF* X *AND* Y | $X - Y$ |
| TIMES *OF* X *AND* Y | $X \otimes Y$ |
| QUOTIENT *OF* X *AND* Y | $X \div Y$ |
| IS X Y | sets internal flag to true if $X = Y$ (equality of arguments for numbers; character by character equality of words; word by word equality of sentences); false otherwise. |
| *IF* YES *THEN* $S_1$, when $S_1$ is some executable statement | execute $S_1$ if internal flag is true; ignore $S_1$ if false. |
| *IF* NO *THEN* $S_1$ | execute $S_1$ if internal flag is false; ignore $S_1$ if true. |
| *IF* WORD? *OF* X *THEN* $S_1$ | executes $S_1$ if X is a word. |
| *IF* *SENTENCE*? *OF* X *THEN* $S_1$ | executes $S_1$ if X is a sentence. |
| *IF* NUMBER? *OF* X THEN $S_1$ | executes $S_1$ if X is a number. |
| TO NAME *OF* < X > AND < Y > | begins definition of a function named "name" and whose formal parameters are X and Y. |
| RETURN X | exit from current function with value X. |
| END | complete definition of function and insert RETURN ' ' in the code for safety's sake. |
| GO *TO LINE* N | branching statement to be used inside of user-defined functions. |
| CALL *THING* X *NAME* Y | associates the name produced by evaluating Y with the value produced by evaluating X. |
| LOGO | reset. |
| ERASE name | erase the function named "name." |
| TRACE | turn on trace for all user-defined functions. |
| UNTRACE | turn off the trace. |
| PRINT X | print the value of X on the user's teletype. |

# A touch sensitive X-Y position encoder for computer input

*by* A. M. HLADY

*National Research Council*
Ottawa, Canada

## INTRODUCTION

Any input device used in conjunction with a computer controlled display for interactive information exchange between man and computer must function as a position encoder. Input devices for handling two dimensional positional information can be grouped into two general types, one type encoding absolute positions and the other encoding changes in position.

Devices accepting absolute positions rely on a direct mapping of positions from an input surface to a display surface. The input surface is usually a flat plate or tablet on which positions are indicated with a movable hand held stylus. One consideration in developing a device of this type is the location of the input surface with respect to the display surface. The mapping relationship between surfaces is simplified for the user to the extent of being instinctive if the two surfaces are coincident. If the input surface is superimposed on the display surface with a finite separation, the user has to cope with the problem of parallax. A transparent input surface and a one to one mapping scale are implicit in these two arrangements. A third possibility is that the two surfaces are in different physical locations. This makes it necessary for the user to rely on a visual feedback process by observing the mapping of his selected position in relation to the desired position and then modifying his selection to decrease the difference.

The stylus used for indicating positions on the surface is typically an active one which contains a signal sensor, as for example, in the RAND Tablet,[1] or a signal radiator, as in a magnetically coupled device described by Lewin.[2] The stylus must be large enough to accommodate the necessary components, and, in addition, present devices require a cable connecting the stylus to the console for signal transmission. his makes some active styli difficult to use with dexterity.

Input devices for encoding position increments do not have separate input surfaces, and their operation depends entirely on visual feedback from the display surface. This type of device consists of a mechanical assembly having at least two degrees of freedom, such as a joy-stick or track-ball, which can be manipulated to indicate changes in the position of a cursor displayed on the screen.

### *Touch sensitive overlay*

Work on the device described in this paper began with several primary objectives which are related to the considerations outlined above. These objectives are:

1. The device must encode absolute positions indicated by the user.
2. The input surface must be as close as possible to the display surface.
3. Positions are to be indicated with a passive stylus, including a human finger.

The first two objectives ensure that the relationships between the positional information that the user must provide and the information he observes on the screen are fundamental ones. This reduces the time and mental effort expended, especially when the device is used for item selection, that is the selection of a sub-set from a set of items shown on the display surface.

Assuming that the first two objectives are met, the third allows one to select items or positions on the screen merely by pointing at them with a finger. Because pointing with a finger is man's most natural method of indicating selection, a touch activated device creates a minimum of distraction for the user. In fact, an ideal implementation of the three objectives listed above would result in an input device that was apparent to the user in function rather than in substance.

Admittedly, the human finger is a rather coarse stylus but the resolution attainable is sufficient for many types of manual information entry. The words or phrases displayed for selection in an information retrieval system could be in a format suitable for this type of input technique. If a conventional keyboard is used in conjunction with the display terminal, a touch activated display overlay reduces the time spent in going from keyboard to display by eliminating the intermediate step of picking up a stylus. In addition, a portion of the display screen could be used as a touch sensitive keyboard with dynamic computer control of the associated key functions. The apparent simplicity, both physically and functionally, of this type of input device is a significant advantage if the user is a young child communicating with a computer-assisted instruction system.

For information entry requiring more resolution than one can obtain with a finger, a suitable passive stylus could resemble an ordinary pencil with its convenient size, light weight, and freedom of movement.

One touch sensitive device[3] that has been developed for use with a CRT consists of a number of wires terminating at the front surface of the display tube. Each wire forms the arm of an AC bridge which is unbalanced by body capacitance. A second device, developed by Control Data Corporation, has a series of translucent, touch-activated strips in front of a CRT display.

The approach taken in our case was to use an echo ranging technique with elastic surface waves. Echo ranging with pulsed ultrasonic surface waves has been applied successfully for a number of years in the field of flaw detection for structural materials. The propagation delay of ultrasonic elastic waves has been used as the basis for graphic input devices for a computer. However, these devices do not employ echo ranging and consist basically of fixed sources or radiators with the sensor in a movable stylus. One of these, developed by Woo at IBM,[4] also uses surface waves on a glass plate. The Lincoln Wand[5] provides a three dimensional input capability by using ultrasonic waves propagating in air.

In the device developed at NRC, the radiator and sensor are physically the same piezoelectric transducer which is electrically switched between the driving circuitry and the echo receiving circuitry. Pulse modulated surface waves are produced on a transparent glass plate, and any object contacting the surface reflects some of the wave energy back to the source. The distance from the radiator/sensor to the target is proportional to the time between the radiator pulse and the reception of the echo pulse.

### *Surface wave characteristics*

An elastic surface wave can be represented mathematically as a combination of inhomogeneous longitudinal and transverse waves. This is exemplified by the particle displacements for a surface wave. The particles describe elliptical orbits with the major axis perpendicular to the surface and the minor axis parallel to the direction of propagation, corresponding to the transverse and longitudinal components respectively.

The particle displacements decrease exponentially with depth into the material, the depth decay factor being a function of the wavelength and the material. For glass, the wave energy at a depth of one wavelength is only about three percent of its value at the surface. A practical implication of this result is that, to a close approximation, a plate several wavelengths thick appears as the solid half-space necessary for true surface wave propagation.

Waves on the free surface of a solid half-space, which are also known as Rayleigh waves, are not dispersive and their phase velocity depends only on the properties of the material on which they are propagating. For plate glass the velocity is 10,400 ft/sec.

The amplitude of all elastic waves decreases with distance from the source through three mechanisms—beam divergence, scattering, and absorption. Because a surface wave is essentially a two-dimensional phenomenon, the decrease in amplitude due to beam divergence is proportional to $1/\sqrt{r}$, compared to $1/r$ for spatial waves, where r is the distance from the source. The attenuation due to scattering and absorption is related to that of spatial waves, with the attenuation factor being approximately proportional to frequency in the ultrasonic range. The attenuation coefficient of plate glass measured at 8 MHz is 0.40 nepers/inch.

An interesting property of surface waves is their ability to propagate along curved surfaces. If the radius of curvature is large with respect to the wavelength, there is only a slight change in attenuation and velocity. This property makes it possible to employ the echo ranging principle described to produce a device which uses the curved front face of a CRT as the input surface, reducing parallax to a practical minimum.

### *Echo ranging parameters*

All systems using echo ranging for target location have similar design parameters. Although considerable effort has gone into the refinement of echo ranging techniques for radar and sonar, the additional complexity and cost of such developments as signal correlation makes them impractical for this application.

For two dimensional space, the stylus location can be determined by measuring its distance from two fixed points or its normal distance from two fixed lines. The latter method was chosen and implemented by alternately scanning the surface in orthogonal directions using linear transducer arrays fixed at the edges of a square plate. This method can provide the stylus location directly in terms of x-y coordinates without additional computation. The line reference method also avoids the problem of edge reflections obscuring valid echoes. Furthermore, with the large beamwidths needed in the first method, it is difficult to achieve an adequate surface wave power density at frequencies in the megahertz range.

The choice of plate material was limited by the transparency requirement. Ordinary plate glass was found to be satisfactory although its attenuation coefficient is higher than that of fused quartz and some optical glass. All the glass tested had several surface flaws per square foot but most of these were shallow enough to be eliminated by localized hand grinding and polishing. The plate size was chosen to provide a usable surface of 10 × 10 inches.

Factors involved in the choice of carrier frequency include the positional resolution, the surface wave attenuation, the radiator beamwidth, the gain in reflected energy for a given target size, and the availability of piezoelectric transducers. A carrier frequency of 8 MHz was chosen for the initial device with the corresponding wavelength on glass being about 0.015 inch.

### *Radiator/sensor development*

One of the most efficient and convenient ways of generating surface waves at frequencies in the low megahertz range is by the mode conversion of a longitudinal spatial wave. This occurs when a longitudinal wave is incident on an interface between two solid

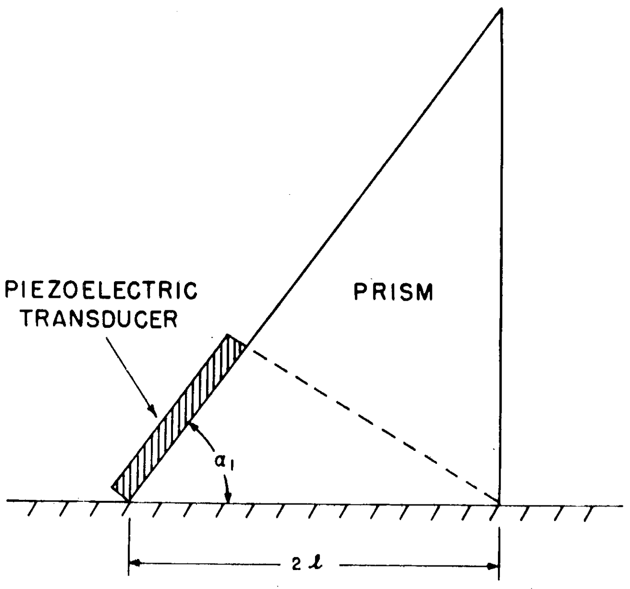


Figure 1—Surface wave radiator/sensor

materials with an angle of incidence large enough that total internal reflection occurs, and no energy is refracted into the second material. In order that the boundary conditions remain satisfied at the interface for this case, inhomogeneous longitudinal and transverse waves are produced in the second material. In ther words, a surface wave is generated.

A practical implementation of this, shown in Figure 1, consists of a thickness mode piezoelectric transducer mechanically coupled to a solid prism. Maximum surface wave output occurs for a prism angle, $\alpha_1$, such that the spatial period of the surface perturbations corresponds to the wavelength of the resultant surface waves at the frequency of the incident wave. That is, when

$$c_L = c_S \sin\alpha_1$$

where $c_L$ is the longitudinal wave velocity in the prism,

and $c_S$ is the surface wave velocity.

For this optimum angle to be real, the prism material must be chosen so that $c_L < c_S$. One of the commonly available materials that meets this velocity requirement for generating surface waves on glass is an acrylic resin such as Plexiglass or Lucites.

The same configuration also makes an efficient surface wave sensor. In this case, incident surface waves excite spatial waves in the prism with an angle of propagation determined by the velocity ratio. When the same transducer is used for both sending and receiving, the energy that was internally reflected within the prism during the send interval appears as clutter or noise during the receive interval. Although this excess energy is gradually absorbed by the prism material, its effect can be reduced by modifying the prism shape and coating it with an absorbent material. For the transducers actually constructed, the first two inches of range could not be used because of the clutter.

The piezoelectric transducers are made of a lead zirconate-lead titanate ceramic having a thickness mode electro-mechanical coupling coefficient of 0.66. This material is relatively good for energy transformation in both directions. The bandwidth and mechanical output power of a piezoelectric transducer are related to the mechanical impedance of the materials to which it is coupled. After some experimentation with quarter wave impedance matching transformers and various backing materials, it was decided to sacrifice bandwidth for sensitivity by using air-backed transducers bonded directly to the prism. The result was a radiator fractional bandwidth of 20 percent. The parallel components of the electrical input impedance for a small test array constructed in this way are shown in Figure 2.

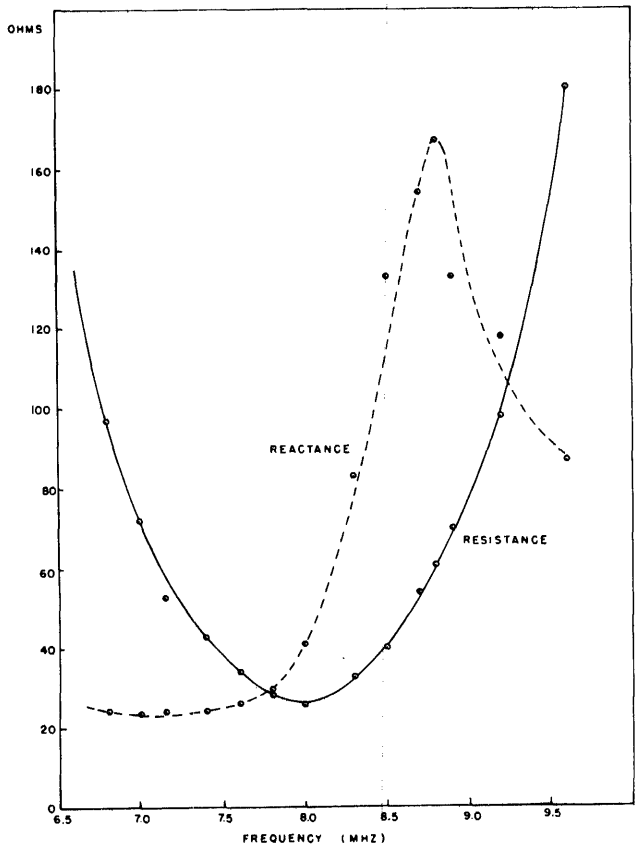


Figure 2—Parallel impedance components for a series connected array of four 1/2 × 1/4 inch transducers

For an 8 MHz pulse modulated signal with a 1.6 MHz bandwidth, the minimum resolvable stylus movement should be about 0,04 inch. As will be explained later, this resolution was attained but unusable in the first device constructed.

*Array design*

The method of target location being used requires a line source of waves having uniform amplitude and phase across a ten inch width. To combine separate radiator elements into a linear array with the desired characteristics, the radiation pattern of individual elements must be known. An expression for the directivity characteristics of a prism type of radiator has been derived,[6] and it yields results similar to the sin x/x function for spatial radiators. Figure 3 compares values computed for an 8 MHz radiator using this expression with experimentally measured values.

For practical plate dimensions and transducer sizes, the usable surface area lies in the far-field region of the individual elements but in the near-field region of the overall array. By computing the response for various linear array configurations, a radiator width of 0.465 inch, and a spacing of 0.565 inch, were selected.

After the arrays were assembled and tested, the measured radiation pattern was more irregular than the computations indicated. This discrepancy was attributed to the variation is spacing, orientation, and bond characteristics due to assembly tolerances and the variations in transducer sensitivity. The gaps in the pattern were sufficiently large and numerous that it was necessary to add a second set of arrays on the opposite sides of the plate. These are offset with respect to the first so that the beams from opposite arrays are effectively interleaved. The arrays are energized sequentially to avoid mutual interference.

The maximum two-way propagation time for a ten inch usable surface and a two inch buffer zone is about 200 $\mu$sec. Therefore, even with four separate arrays, the sampling rate can be greater than 1 KHz, which is more than adequate to follow normal stylus motion.

*Electronic circuitry*

The signal processing circuitry consists of a radiator driver, an electronic switch, and an echo receiver. The

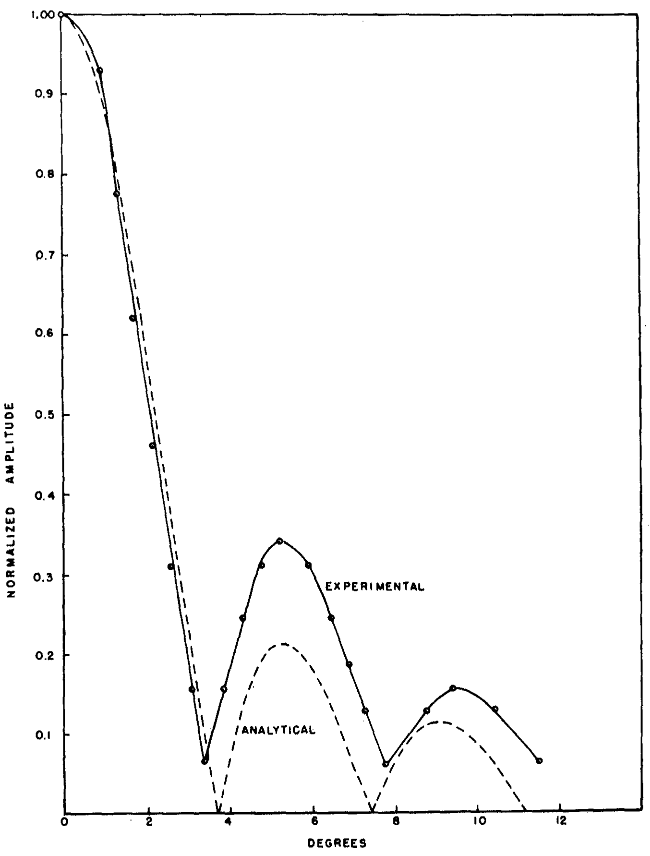


Figure 3—Directivity pattern for a surface wave radiator at 8 MHz with 0.23 inch width

timing circuitry digitizes the signal propagation time, and the control logic maintains the correct operating sequence. Figure 4 shows how these components are interconnected.

The radiator driver and the arrays are matched to 50 ohms allowing them to be connected with standard coaxial cable. The diode switch, with a four-pole double-throw action, permits the four arrays to be multiplexed into a single driver and receiver, and it also isolates the receiver during the driver pulse. The echo receiver consists of an RF amplifier followed by a demodulator and a threshold detector. The receiver gain is electronically swept during each scan to compensate for the signal attenuation with range. A range gate rejects echoes originating outside of the designated area. Figure 5 shows the demodulator and threshold detector outputs for a single scan. The signal at the centeris the echo from a finger touching the glass.

Echo timing is performed by a free running counter. Both up and down counting are required to digitize scans originating at opposite sides of the input surface. The coordinate grid is considered to have X and Y axes coincident with the edges of the usable surface, the origin being in the lower left corner. Adjustments on the range gates and counting circuitry allow the size and position of the coordinate grid to be varied slightly to permit registration with the grid of an associated display device.

The control circuitry allows two modes of operation: a continuous mode and a discrete mode. In the continuous mode, a Data Ready pulse signals the comput-

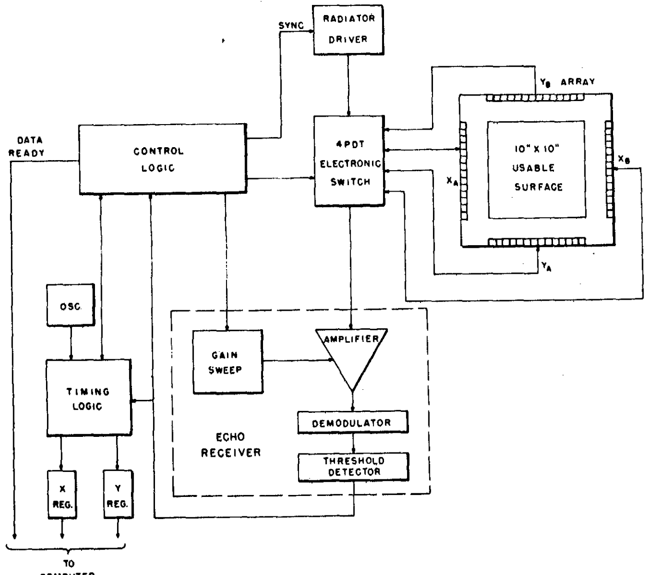


Figure 4—Position encoder block schematic

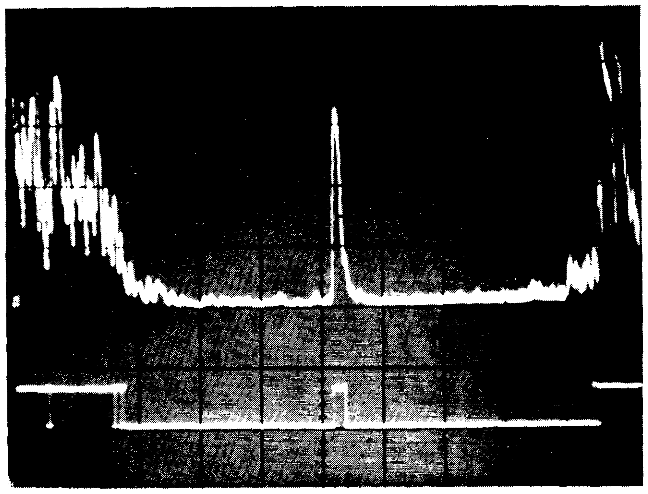


Figure 5—Echo receiver response
Vertical: Upper 0.5 v/div., Lower 5.0 v/div.
Horizontal: 25 $\mu$sec/div.

er for every set of coordinates generated while stylus contact is maintained. In the discrete mode, on the other hand, only the location of the initial contact is transferred to the computer. The stylus must be lifted and repositioned to initiate another data transfer. The discrete mode significantly reduces the amount of data that must be handled without degrading the response time when the device is being used for item selection purposes only.

In applications such as CAI which require a cluster of computer terminals in one location, it becomes feasible to time-share the electronic circuitry among several terminals, thereby decreasing cost per unit.

*Device performance*

The complete device is shown in Figure 6 with a static display card behind the glass for demonstration purposes. It has been interfaced with a Digital Equipment Corporation PDP-8 computer for testing and evaluation.

Tests have shown that stylus movements of 0.04 inch could be resolved, which corresponds to the calculated value mentioned earlier. However, it was found that a contact area approximately ¼ inch in diameter is necessary to ensure operation anywhere on the $10 \times 10$ inch surface. The contact area must be as large as that to bridge the regions of low sensitivity which result from the irregularities in the surface wave radiation pattern. This means that even though the device has an inherent positional resolution of 0.04 inch, the usable working resolution is considerably lower.

When using the device with a finger, a pressure of only a few ounces is adequate for operation over most of the surface. In a few places, the pressure has to be increased to enlarge the contact area sufficiently. In the former case, pointing with a finger to items displayed behind a seemingly ordinary glass plate is quite natural, and, except for the parallax, a person can make use of the device without consciously being aware of its presence.

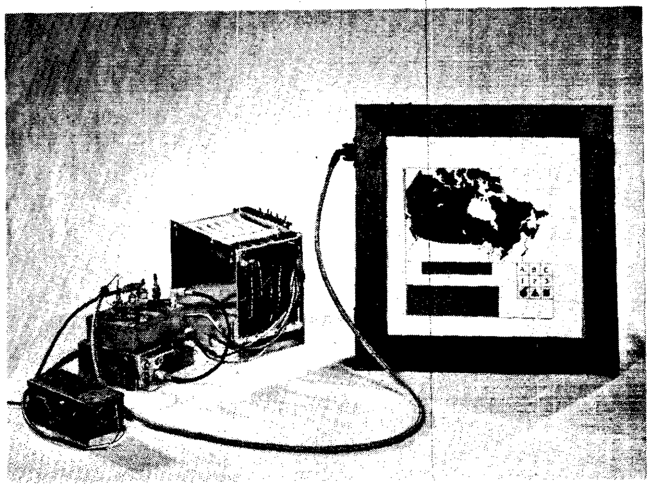


Figure 6—Touch sensitive position encoder

The position encoding is accurate and linear to about 0.5 percent. This figure takes into account the variation in wave velocity due to temperature change and material inhomogeneity, nonlinearity of the radiated wavefront, and the stability of the timing circuits.

Because scratches and marks on the glass produce small echoes which contribute to the background noise level in the receiver, some care must be used to keep the surface clean. The accumulation of fingerprints on the glass also contributes to the background noise. However, this is not a serious problem when the device is used with reasonably clean hands.

The initial device as described has served to demonstrate the feasibility of using surface wave echo ranging as the basis for a touch-sensitive position encoder. The experience gained in constructing and testing the device has been useful in determining where improvements are needed and how they should be implemented. Further computations indicate that a more sophisticated approach to the array design and assembly should improve the radiation pattern uniformity and thereby reduce the present disparity between the minimum contact size and the inherent resolution. Tests have been shown that lowering the carrier frequency to about 4 MHz should increase the signal-to-noise ratio of usable stylus echoes by decreasing the signal attenuation and lowering the sensitivity to surface contamination. The overall consequences of these changes will be to improve the performance with medium and low resolution styli and also to simplify the circuitry, and hence reduce the cost, by using two arrays instead of four. Work is progressing on the construction of a device which incorporates the improvements described.

## REFERENCES

1 M R DAVIS T O ELLIS
*The RAND tablet: A man-machine communication device*
AFIPS FJCC Proc Vol 26 325 1964

2 M H LEWIN
*A magnetic device for computer graphical input*
AFIPS FJCC Proc Vol 27 831 1965

3 E A JOHNSON
*Touch display: A novel input/output device for computers*

Electronics Letters Vol 12 1964 Vol 13 1965
4 P W WOO
*A proposal for input of hand drawn information to a digital system*
IEEE Trans on Electronic Computers EC-13 609 1964
5 L G ROBERTS
*The Lincoln wand*
AFIPS FJCC Proc Vol 28 223 1966
6 I A VIKTOROV O M ZUBOVA
*Directivity diagrams of radiators of Lamb and Rayleigh waves*
Soviet Physics-Acoustics Vol 9 1962 Vol 138 1963
7 I A VIKTOROV
*Rayleigh waves in the ultrasonic range*
Soviet Pysics-Acoustics Vol 8 1962 Vol 118 1962

# A queueing model for scan conversion

*by* T. W. GAY, JR.

*IBM Systems Development Division*
Kingston, New York

## STATEMENT OF PURPOSE AND EXPECTED RESULTS

The purpose of this paper is to present a queueing model for analyzing a video scan converter (VSC). The system analyst constantly strives for quicker methods, parallel approaches, and more accurate results. Queueing theory is generally useful in the first and second of these categories. How then does the analyst develop a queueing model of a VSC in the hardware development and design stage?

The model is constructed through study of the internal functioning of the VSC and a queueing model is then developed which functions analogously with it. The queueing model developed for the VSC was an extension and adaptation of the known queueing model called "the machine interference queueing model." (See first section for an explanation).

The general machine interference queueing model was extended and modified to permit the servicing of multicharacter conversions in lieu of single character conversions.

The results are presented in the first two sections of this paper.

## INTRODUCTION AND EXPLANATION OF THE VIDEO SCAN CONVERTER

Queueing analysis is a recent branch of probability theory which studies the characteristics and effects of congestion in systems subject to random flows. The system under study may be a supermarket, a busy airport, or a real-time message processor. Ideally, the behavior of each of these systems could be represented in mathematical terms, the common elements identified, and the appropriate analysis applied to determine the expected effects of various modes of operation. Practically, however, no such extensive analysis can be carried out. This is due in part to the lack of complete knowledge of the system specifications at the time analysis is required. But more important is the present limitation of the mathematics.

The function of the video (analog) scan converter is to effect the conversion of characters which have been generated by a computer in directed beam format into a video scan format. In fulfilling this function, a video scan converter ordinarily "paints" character(s) on the face of a cathode ray tube and "converts" their image by scanning the image with a Vidicon. The directed beam character appears to be painted on with no presence of dots (or scanning lines). The painted image is converted to a video scan character and is composed of horizontal dots conforming to the character shape. The smooth painted character has now become a configuration of dots close enough together so that the eye perceives an entire character(s).

One video scan converter is normally used to service a group of video displays. If a keyboard is attached to a video display, then the operator can enter keystrokes thru the keyboard into the computer. The keystroke(s) are converted to the video scan format and appear on the operator's display screen. If characters appear on the display, one by one, this is called "single character conversion", a subject not discussed in this paper. However, characters frequently appear on the display screen in groups due to (batches) because of high traffic. This paper assesses this multiple character conversion phenomenon.

*A queueing model for the video scan converter*

Explanation of the general case "machine interference" queueing model, with development of associated equations. (Reference: Feller, W., *An Introduction to Probability Theory and its Applications*, 2nd edition, New York: J. Wiley and Sons, 1957, Pages 416-418.)

The machine interference model is a general class of queuing models. We are here specifically interested in the "Machine Servicing With Single Serviceman."

This model has a finite number of customers arriving randomly at a single server. It was originally applied in Swedish industry to determine how many machines (customers) one setup man (server) could tend without undue waiting delays resulting from several machines requiring service at the same time.

Assume there are "m" identical machines assigned to one serviceman. Each machine is in one of two states.

1. "up" (operating)
2. "down" (requiring service)

When a machine goes "down", it joins the queue for the serviceman. If the serviceman if free, he immediately begins to service the machine. If he is busy, the machine must wait for service. The queue (waiting line) is organized on a "first-in, first-out" basis. The design is shown in Figure 2.

To obtain the only readily available solution, the following assumptions are made:

1. Service Time for all machines is expotentially distributed with mean time, "$T_s$".
2. The "up" time for each machine expotentially distributed with mean time, "$T_a$".

These assumptions result in worst-case answers if the actual service and "up" time distributions are more regular.

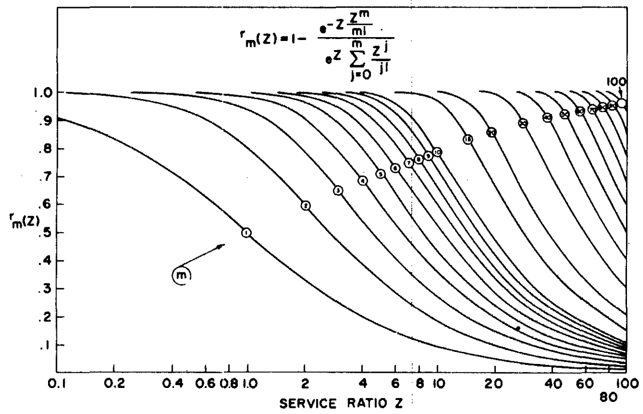


Figure 1—Poisson ratio function vs. service ratio

Since there is a "fixed" number of customers, we can readily see that the arrival rate of the machines to the service queue is proportional to the number still operating. If all machines are in the queue, the arrival rate is reduced to zero. Because of this "captive audience" characteristic, the system has a built-in limiting effect and cannot become unstable (no infinite number of customers in the queue). For a relatively efficient machine, the mean operating time, $T_a$, is comparatively large compared to mean servicing time, $T_s$. The ratio of these two values is termed here the "service ratio", z.

$$z = \frac{T_a}{T_s} \tag{1}$$

If $P_k$ denotes the probability that K machines are "down", then let $P_0$ denote the probability that all machines are operating and the serviceman is idle. No machines are in the waiting line nor being serviced. $P_0$ is the probability which represents the fraction of time the serviceman is idle. Thus, $1 - P_0$ is the frac-

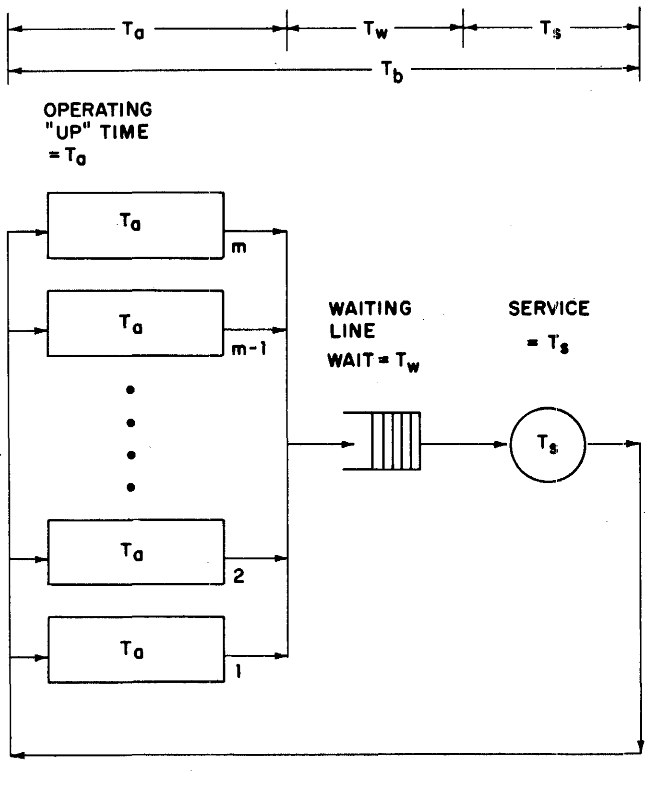


Figure 2—Model of machine servicing with single serviceman

tion of time the serviceman is busy, and can be called the server utilization.
Hence:

$$P_k = \frac{e^{-z}\left(\frac{z^{(m-k)}}{(m-k):}\right)}{e^{-z}\sum_{j=0}^{m}\left(\frac{z^j}{j!.}\right)} \qquad (2A)$$

Where $P_k$ is the probability that k machines are "down" Equation 2A is the ratio of two Poisson expressions, both obtainable from Poisson tables, and is known as the truncated Poisson distribution. If $K = 0$, then Equation 2A would give the probability of *no* machines in the service queue. If $1 - P_0$ is server utilization, then substituting $k = 0$ into Equation 2A and subtracting it from 1 gives:

Server Utilization =

$$(1 - P_0) = 1 - \left[\frac{(e^{-z})\left(\frac{z^m}{m!}\right)}{(e^{-z})\sum_{j=0}^{m}\left(\frac{z^j}{j!}\right)}\right] \qquad (2B)$$

For convenience, this function has been plotted in Figure 1, "Poisson Ratio Function versus Service Ratio." Given $T_a$ and $T_s$, z can be calculated using Equation 1. Given m, the number of individual queues, $r_m$ (z) can be found at the intersection of z and m and its value read from the "y" axis, Figure 1. Note that $r_m$ (z) denotes server utilization.

For each machine, a breakdown is followed by a wait for service, a service time, and an operating time until the next time it has a breakdown. In equation form:

$$T_b = T_w + T_s + T_a$$

Where:

- $T_b$ – is the mean time between breakdowns per machine
- $T_w$ – is the mean time waiting to be serviced per machine
- $T_s$ – is the mean time to service a "down" machine
- $T_a$ – is the mean time a machine is "up", (operating)

The mean rate of machine breakdown is $1/T_b$. Since there are m machines, the total mean rate of machine breakdowns entering the service queue is $m/T_b$. Each breakdown requires a service time $T_s$. Therefore, the server utilization, $r_m$ (z), must be:

$$r_m(z) = \frac{mT_s}{T_b} \qquad (4)$$

But if $r_m$ (z) is already known thru use of Figure 1, then Equation 4 can be solved for $T_b$:

$$T_b = \frac{m\,T_s}{r_m(z)}; \quad \text{where } T_b \text{ is mean time between breakdowns.} \qquad (5)$$

By further examination of Equation 3, it can be seen that the mean time a machine stays in the "down" state is:

$$T_w + T_s = T_b - T_a = \left[\frac{m\,T_s}{r_m\,(z)} - T_a\right] \qquad (6)$$

A correlation which will be made later is that $T_w + T_s$ is sometimes referred to as average response time, $T_r$. A useful alternate form to Equation 6 is:

$$T_w + T_s = \left[\frac{m}{r_m\,(z)} - z\right]T_s \qquad (7)$$

Since $T_a = z\,T_s$, and substitute for $T_a$ in Equation 6. The mean number of down machines in the waiting line, not including the one in service, is given by:

$$L_w = \sum_{k=1}^{m}(k-1)\,P_k = m - (z+1)\,r_m\,(z) \qquad (8)$$

The mean number of all "down" machines, including the one in service, is given by:

$$L_q = \sum_{k=1}^{m} k\,P_k = L_w + r_m\,(z) = m - z\,r_m(z) \qquad (9)$$

Where, in Equations 8 and 9 above:

- $P_k$ is the probability that k machines are down k is the number of machines "down"
- m is the total number of machines in the systems and is a constant
- z is the ratio of the machine "up" time to the machine service time.
- $r_m$ (z) is the server utilization

The proportion of time that a machine spends in the "down" state is found by dividing Equation 6 by $T_b$:

Prob (machine K and only machine k is "down") $= (1 - z/m\, r_m(z))$ (10)

**Example**

Suppose that eight machines are tended by one serviceman. The mean operating time of a machine is 380 seconds, and mean service time is 34.5 seconds. Assume that both operating and service times are exponentially distributed. Determine the operating characteristics of this system.

The service ratio is $z = \frac{T_a}{T_s} = \frac{380 \text{ seconds}}{34.5 \text{ seconds}} = 11$

a. What is the serviceman's utilization?
Using Figure 1 ,with $z = 11$ and $m = 8$
$r_m(z) = r_8\ (11) = .62 = 62\%$, which is the serviceman's utilization

b. What is the average number of "down" machines?
Using Equation 9, with $m = 8$, $r_8\ (11) = .62$, and $z = 11$ $L_q = 8 - 11\ (.62) = 8 - 6.82 = 1.18$, which is the average number of machines "down" and are located in the waiting line or in service.

c. What is the average time a machine spends in the "down" state?
Using Equation 6, with $m = 8$, $r_8(11) = .62$, $T_a = 380$ seconds, $T_s = 34.5$ seconds.

$$(T_w + T_s) = \frac{mT_s}{r_8(11)} - T_a = \frac{8\ (34.5) \text{ sec.}}{.62} - 380 \text{ sec}$$

$(T_w + T_s) = 445$ sec. $- 380$ sec. $= 65$ seconds, which is the average time a machine is "down."

d. What fraction of the total time is a machine in the "down" state?
Using Equation 10, with $m = 8$, $r_8\ (11) = .62$, $z = 11$. Prob = Fraction of total time = $(1 - 11/8\ (.62)) = .15 = 15\%$

Let us consider the same example again, the one we have just used to determine operating characteristics. To illustrate the practicality of the case of "Machine Servicing With a Single Serviceman" let us transform the example by considering the analogies we wish to introduce.

| ITEMS FOUND IN ORIGINAL EXAMPLE | ANALOGOUS ITEM NOW REPLACING THE ORIGINAL |
|---|---|
| 8 machines, $m = 8$ | 8 independent sources for incoming data; $m = 8$ |
| one serviceman, $n = 1$ | one service facility, $n = 1$, required to service all eight sources of data (the video scan converter) |
| 380 seconds = the mean operating time, $T_a$, per machine (time frame is immaterial) | 380 milliseconds = the average inter-arrival time, $T_a$, between characters coming from any one source of data |
| 34.5 seconds = the mean service time, $T_s$, per machine (time frame is immaterial) | 34.5 milliseconds = the average service time per character of input from any source of data, $T_s$ |
| serviceman's utilization $r_m$ (z) | utilization of video scan converter (servicer), $r_m$ (z) |
| down machines, $L_q$ | total characters waiting or being serviced in the system, $L_q$ |
| the average time a machine spends in the "down" state = $(T_w + T_s)$ | the average time a character spends waiting for and receiving service = $(T_w + T_s)$, response time. |
| $T_b$ is the average time between "breakdowns" per machine, and is the sum of $T_a$, $T_w$ and $T_s$ | The average time interval between services of a specific queue; $1/T_b$ is the average number of queues serviced during this time interval |
| Prob (machine K is in the "down" state) = fraction of the total time a machine is in the "down" state | Prob (that any character in the system is waiting or is being serviced) = fraction of the total time any character in the system is waiting or is being serviced |

**Transformed example continued**

The service ratio is $z = \frac{T_a}{T_s} = \frac{380 \text{ ms}}{34.5 \text{ ms}} = 11$

a. What is the average scan converter utilization? Using Figure 1, with z = 11, m = 8

$r_m(z) = r_8(11) = .62 =$ *62% utilization*

b. What is the average number of characters in the system?

Using Equation 9, with m = 8, $r_m(11) = .62$, $z = 11$ $L_q = m - z\, r_m(z) = 8 - 11(.62) = 8 - 6.82 = 1.18$ characters on the average are in the system waiting or being serviced.

c. What is the average response time per character?

Using Equation 6, with m = 8, $r_8(11) = .62$ $T_a = 380$ ms, $T_s = 34.5$ ms

$$(T_w + T_s) = \left(\frac{m\,T_s}{r_8(11)} - T_a\right)$$

$$= \left[\frac{8\,(34.5 \text{ ms})}{.62} - 380 \text{ ms}\right]$$

$(T_w + T_s) = [445 \text{ ms} - 380 \text{ ms}] = 65$ ms, average response per character

d. What fraction of the total time does a character spend waiting for or being serviced?

Using Equation 10, with m − 8, $r_8(11) = .62$, z = 11

$$\text{Fraction of total time} = (1 - z/m\,(r_8(11))$$
$$= (1 - 11/8\,(.62))$$
$$= .15 = 15\%$$

*Extension and adaption of the general case "machine interference" queueing model to permit multiple character updates per service cycle.*

Consider now that we wish to adapt the single character update model to one which is capable of representing a multiple character update. Specifically we mean the ability to service "N" characters coming from the same source and residing in the same queue in the same 34.5 milliseconds service cycle. In effect, the service time per character reduces to:

$$T_s' = \frac{T_s}{N} = \frac{34.5 \text{ millisecond}}{N \text{ Characters}} \tag{11}$$

We are especially interested in the response time, $T_r$, since this provides a measure of "machine responsiveness" to a keyboard operator entering a character stream into the system. $T_r$, is meant to be the average response time per character, since the response time for the first character will be longer than that of the last character awaiting service from the same source.

As with our previous model, our service ratio is defined as:

$$z = \frac{T_a}{T_s} \tag{12}$$

Also using Figure 1, the server utilization, $r_m(z)$, can be found at the intersection of z and m, and its value read from the "y" axis.

$$\text{Let } T_r^* = (T_w + T_s) = \left[\frac{m}{r_m(z)} - z\right] T_s \tag{13}$$

During the time interval between services, $T_b$, the number of characters which arrived at a specific queue is:

$$N = \frac{T_b}{T_a} = \left[\frac{(T_r^* + T_a)}{T_a}\right] = \left[\frac{T_r^* + 1}{T_a}\right] \tag{14}$$

Where N is the character contents of an individual queue and is the average number of characters services as a "batch".

Referring to Figure 3, in a typical multiple character service there are N characters and N − 1 time intervals, $T_a$, between the characters. $T_a$ is the average interarrival time of the incoming character stream. As the wait time becomes longer more characters arrive at the individual queue, awaiting service simultaneoulsy with the first character in the queue. When the queue is serviced, all characters residing in the queue at that point in time are serviced in the same constant service time of 34.5 millisecond for all N of them. Note that the service time per character has been effectively reduced to 34.5 ms/N characters.

The response time per character must reflect, how-

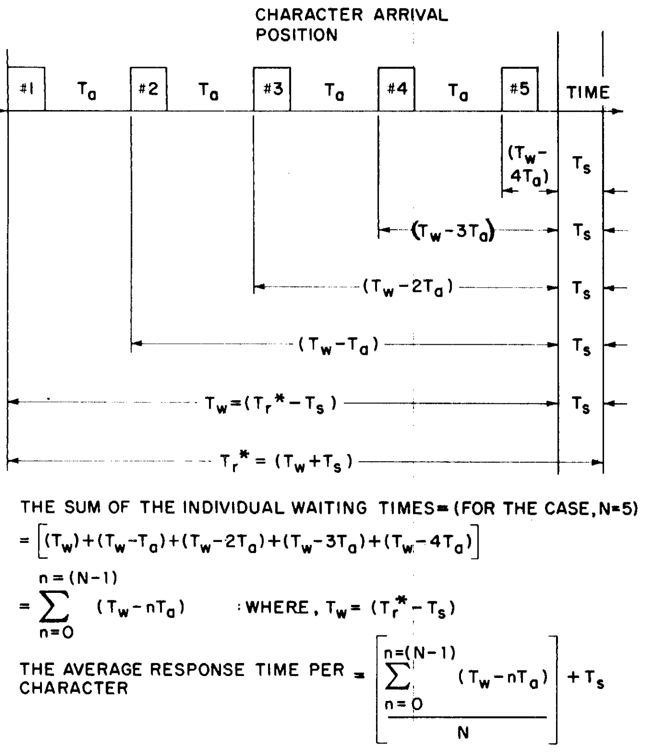

CHARACTER ARRIVAL POSITION
#1
Ta
#2
Ta
#3
Ta
#4
Ta
#5
TIME
(Tw-4Ta)
Ts
(Tw-3Ta)
Ts
(Tw-2Ta)
Ts
(Tw-Ta)
Ts
Tw=(Tr*-Ts)
Ts
Tr* = (Tw+Ts)
THE SUM OF THE INDIVIDUAL WAITING TIMES = (FOR THE CASE, N=5)
= [(Tw)+(Tw-Ta)+(Tw-2Ta)+(Tw-3Ta)+(Tw-4Ta)]
= Σ n=0 to n=(N-1) (Tw-nTa) : WHERE, Tw = (Tr*-Ts)
THE AVERAGE RESPONSE TIME PER CHARACTER = [Σ n=0 to n=(N-1) (Tw-nTa) / N] +Ts

Figure 3—Time profile of 5 characters, ($N = 5$), awaiting service in the 1th queue and the summation of total time, $T_wN = 5$

ever, that all N characters did in fact require 34.5 ms each while they were serviced as a "batch"

The concept used to obtain the average response time per character was to first separate the waiting time from the service time portion found in the first character response time:

$$T_w = (T_r^* - T_s) \tag{15}$$

Where $T_w$ is the waiting time only of the first character, $T_s$ is the service time of the first character, and $T_r^*$ is the response time for the first character.

The sum of the individual character waiting time (See Figure 3)

$$T_{sum} = \sum_{n=0}^{n=(N-1)} (T_w - nT_a) \tag{16}$$

The average waiting time per character is:

$$T_w \text{ (average)} = \left[ \frac{\sum_{n=0}^{n=(N-1)} (T_w - nT_a)}{N} \right] \tag{17}$$

Since the service time was subtracted out in Equation 15, it must be re-entered so that each and every character in the "bash" is charged with $T_s$. Inserting $T_s$ into Equation 17, gives the average response time per character, $T_r$:

$$T_r = \left[ \frac{\sum_{n=0}^{n=(N-1)} (T_w - nT_a)}{N} \right] + T_s \tag{18}$$

Equation 18 is important since it is the mathematical expression we originally set out to find. The reader should note that $T_r$ is the Overall Average Response Time per Character. The following example should be of interest.

## Example

Assume m = 32 independent sources of input character streams, each assigned to an individual queue. Each queue is serviced on a "first come-first served" priority, as determined by the arrival of the first character to enter the individual queue. Let $T_a = 313$ milliseconds, $T_s = 34.5$ milliseconds. Find $T_r$, the Overall Average Response Time per Character.

Using Equation 12, the service ratio, z, is:

$$z = T_a/T_s = 313 \text{ ms}/34.5 \text{ ms} = \mathit{9}$$

Using Figure 1, the server utilization, $r_{32}$ (9) = *1.0*
Where m = 32, z = 9

Using Equation 13, the first character response time, $T_r^*$, is:

$$T_r^* = \left[ \frac{32}{1.0} - 9 \right] 34.5 \text{ ms} = \mathit{795\ ms}$$

Where m = 32, z = 9, $r_{32}$ (9) = 1, 0, and $T_s$ = 34.5 ms

Using Equation 14, the character contents of an individual queue, N, is:

$$N = \left[ \frac{795 \text{ ms}}{313 \text{ ms}} + 1 \right] = 3.54 \text{ characters}$$

N = 3.54, is the average character content of an individual queue. The characters are serviced once per cycle.

Using Equation 15, the average wait time for the first character, $T_w$, is:

$$T_w = (795 \text{ ms} - 34.5 \text{ ms}) = \mathit{760.5\ ms}$$

Using Equation 16, the sum of the individual character waiting times is:

$$T_{sum} = [(T_w) + (T_w - T_a) + (T_w - 2T_a)]$$

$$T_{sum} = [(760.5 \text{ ms}) + (760.5 \text{ ms} - 313 \text{ ms}) + (760.5 \text{ ms} - 2 \times 313 \text{ ms})]$$

$$T_{sum} = [(760.5 + (447.5) + (134.5)] = \mathit{1{,}342.5\ ms}$$

Using Equation 18, the overall average response time per character, $T_r$, is:

$$T_r = \left[\frac{1{,}342.5 \text{ ms}}{3.54 \text{ Char.}}\right] + 34.5 \text{ ms} = 414.5 \text{ ms per character}$$

As compared to the value obtained with a simulation model, the following is the % difference:

$$\% \text{ Difference} = \frac{(414.5 - 390) \text{ ms}}{390 \text{ ms}} \times 100\% = +\ 6.3\% \text{ Difference}$$

*Presentation of results with comparison to a simulation model*

Table 1 and Figure 4 show the computed and simulated values from the queueing model described in an earlier section and a simulation (GPSS) model respectively. The purpose of simulating the video scan converter was to establish validity of the queueing model results, within a range of + or — 20%.

The argument is valid that error in modeling can exist in:

a. The queueing model
b. The simulation model
c. Both models

Of paramount importance, however, is the underlying principle that the probability is least that *both* models will be in error. As a general rule for confirming validity:

a. Values from both models should be in the same "ball park."
b. Output values from both should increase or decrease as independent variables are changed by like amounts.
c. Produce approximately the same slope of values
d. Provide a reasonable division of positive and negative % "differences" over the range of the model's output.

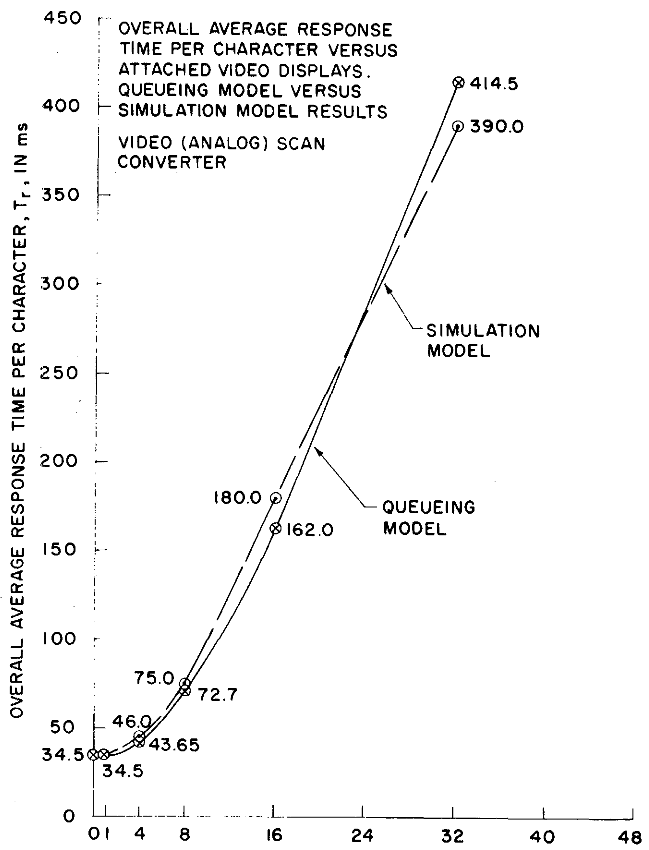


Figure 4—Attached video displays with keyboard, m

Using this as criteria to determine validity the following is my evaluation of the results:

a. The values differed by 10% maximum (at M = 16, the queueing model value for $T_r$ = 162.0 millisecond versus $T_r$ = 180 millisecond for the simulation model.
b. Over the complete range of m = 0 thru m = 32, the values of $T_r$ from both models increased as m was increased a like amount.
c. The slope of $T_r$ values from both models differed over the range of m = 0 thru m = 32. They were:

| *Range* | *Queueing Model* | *Simulation Model* |
|---|---|---|
| m = 0 thru 8, | Slope = + 4.77 | Slope = + 5.06 |
| m = 8 thru 16, | Slope = +11.16 | Slope = +13.12 |
| M = 16 thru 32, | Slope = +15.78 | Slope = +13.12 |
| Totals | +31.71 | +31.30 |

Even though the slopes are somewhat differer they are not appreciably so. It appears t'.e

TABLE I—Computation of values for Figure 4—Overall average response time per character, $T_r$

| m | 1 | 4 | 8 | 16 | 32 | EQUATION |
|---|---|---|---|---|---|---|
| $r_m(z)$ | .10 | .43 | .77 | .995 | 1.0 | FIG.1 |
| $T_r^*$ | 34.5ms | 45.0ms | 83.0ms | 276.0ms | 795.0ms | 13 |
| N | 1.110 | 1.146 | 1.270 | 1.893 | 3.540 | 14 |
| $T_w$ | 0.00ms | 10.5ms | 48.5ms | 241.5ms | 760.5ms | 15 |
| $\sum T_w$ | 0.00ms | 10.5ms | 48.5ms | 241.5ms | 1,342.5ms | 16 |
| $\sum T_w/N$ | 0.00ms | 9.15ms | 38.2ms | 127.5ms | 380.0ms | 17 |
| $T_r=\sum T_w/N+T_s$ | 34.5ms | 43.65ms | 72.7ms | 162.0ms | 414.5ms | 18 |
| $T_r$ (SIM.) | 34.5ms | 46.00ms | 75.0ms | 180.0ms | 390.0ms | SIM. MODEL |
| DIFFERENCE | 0.00ms | -2.35ms | -2.3ms | -18.0ms | +24.5ms | ----- |
| % DIFFERENCE | 0.00% | -5.1% | -3.07% | -10.0% | + 6.3% | 19 |

$$z = \frac{T_a}{T_s} = \frac{278.0\text{ms}}{34.5\text{ms}} = 8 \text{ for } m=1,\ 4,\ 8,\ 16$$

$$z = \frac{T_a}{T_s} = \frac{313.0\text{ms}}{34.5\text{ms}} = 9 \text{ for } m= 32$$

queueing model does not take into account some factor at the low end of m and somewhat over compensates at the high end of m.

d. As shown in Table 1, the maximum negative difference is $-10.0\%$, the maximum positive difference is $+6.3\%$. This is calculated as follows

$$\% \text{ Difference} = \frac{(\text{queueing value, } T_r - \text{simulated,}^{(19)}T_r)}{(\text{simulated, } T_r)} \times 100\%$$

There appears to be a reasonable division between positive and negative % differences.

## CONCLUSIONS

The queueing model as presented, in my opinion, provides a very satisfactory mathematical representation of a video scan converter and better than originally anticipated.

## ACKNOWLEDGMENTS

The author wishes to thank Mr. P. H. Seaman for his help in the form of technical discussion and evaluation; Mr. T. G. Greene for his cooperation in developing the simulation model; Mr. V. L. Hoberecht for his consultation and advice regarding video display systems; and to Mr. D. H. Rumble for his encouragement.

## REFERENCES

1 W FELLER
*An introdction to probability theory and its applications*
John Wiley and Sons 1957 N Y 2nd ed 416-418

2 P H SEAMAN
*Analysis of some queueing models in real time systems*
IBM Data Processing Techniques Manual F20-0007-0 39-42 47-48 (Note: The Poisson ratio function $r_m(z)$ vs. service ratio $z = T_a/T_s$ was obtained from this IBM publication.

3 W FELLER
*An introduction to probability theory and its applications*
John Wiley and Sons N Y 1968 3rd ed 460-468

4 J MARTIN
*Design of real-time computer sytems*
Prentice-Hall Inc Englewood Cliffs N J 1967 396-426

5 D R COX W L SMITH
*Queues*
John Wiley and Sons N Y 1961 general reference to various classes of queueing problems

# Character generation from resistive storage of time derivatives

by MICHAEL L. DERTOUZOS

*Massachusetts Institute of Technology*
Cambridge, Massachusetts

## INTRODUCTION

Recent advances in man-machine communication have stimulated increased interest in techniques and special circuits that generate characters, for graphical and alphanumeric Cathode-Ray-Tube (CRT) display terminals, at the display site. The primary advantage in employing such local character generation is compression of the data that is required to store and communicate a character from the computer to the display—a single binary word of length n is all that is required to instruct the character generator to display one of $2^n$ possible characters. The primary disadvantage of local character generation is display cost, for it is generally considerably less expensive to generate characters from a longer sequence of more elementary commands—for example commands that cause the CRT beam to move right, left, up or down by a minimum resolvable increment. Besides these conflicting costs of data storage and transmission versus local-display generation, several other less tangible criteria such as character stability and fidelity (aesthetics), are instrumental in the design and evaluation of a local character-generation approach.

This paper discusses a character-generation technique which requires, for each character, the storage in a resistive memory of the time derivative functions for the horizontal and vertical CRT deflection signals. The first section of the paper describes specific geometrical primitive segments that can compose a large class of characters and symbols; the choice of such primitives is important, since it affects directly the quality of the displayed characters and the display cost. Also given in this section is a complete list of primitive sequences for the 94-character ASC-II set. The second section of the paper describes a character-generation system that stores the above primitives in a resistor matrix, and uses them to compose desired characters on a CRT display. In the third section, this approach is evaluated and compared to more conventional methods of dot intensification, in terms of cost, speed, and fidelity.

### *Character primitives*

Characters and symbols, generated on CRT displays, are made up of certain elementary graphical segments. *Character primitives* over a character set will be called those segments which are (i) atomic or indivisible to smaller segments, and (ii) sufficient in number and quality to compose within acceptable accuracy every character in that set. At one extreme, the points of a uniformly spaced grid are adequate character primitives (Figure 1a); however, as the number of these points is reduced (Figures 1b and c), it becomes progressively more difficult to recognize the displayed characters. At the other extreme, the set of all characters may be considered itself as a set of character primitives. This set, however, is not very useful, for while it is generally easy to construct a system capable of implementing the primitives of Figure 1, it is considerably more difficult and expensive to implement the primitives at the other extreme. Conversely, it takes only seven bits to specify one of the 94 characters of the ASC-II set, while it takes 49 bits to specify every one of the possible subset of

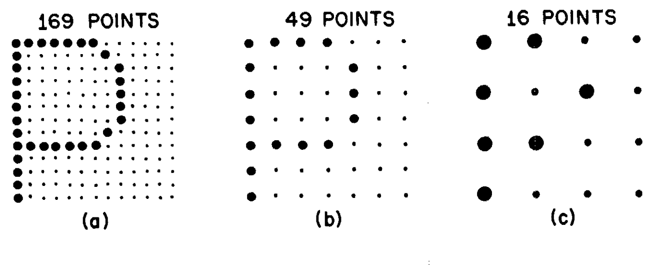


Figure 1—Points as character primitives

dots of Figure 1b. These simple observations on the above two extremes are characteristic of the problems of character generation and of the objectives in the design of an effective character generator—that is the desirability for a small number of primitives which can be economically implemented.

The primitives used in the character generation technique of this paper are continuous strokes which are either (i) straight lines or (ii) so-called "cusps". A straight-line primitive is specified relative to a point P by increments $\Delta_x$, $\Delta_y$ which are real numbers; in our notation each such primitive is denoted, when visible, by $(\Delta_x, \Delta_y)$ or, when invisible by an underscore $\underline{(\Delta_x, \Delta_y)}$. Figure 2a shows two such primitives. The equation of primitive $(\Delta_x, \Delta_y)$ is relative to a coordinate center at point P as follows:

$$\frac{y}{\Delta_y} = \frac{x}{\Delta_x} \text{ for } 0 \leq \frac{y}{\Delta_y} \leq 1,\ 0 \leq \frac{x}{\Delta_x} \leq 1 \qquad (1)$$

where x and y are the horizontal and vertical coordinates of every point on that primitive.

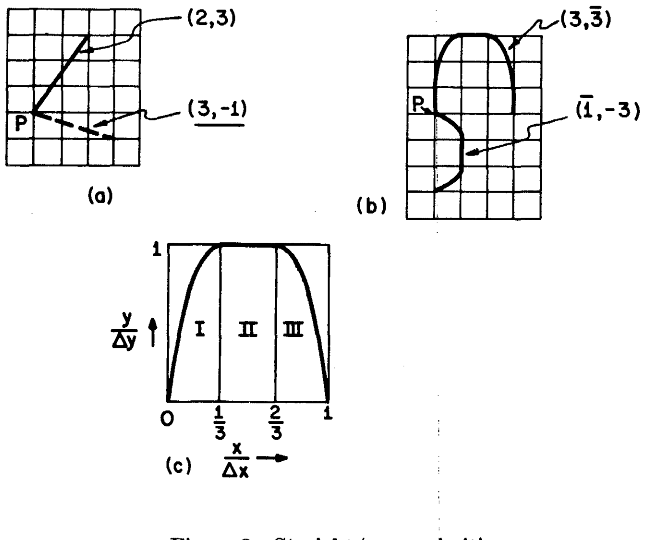


Figure 2—Straight/cusp primitives

The cusp primitive, on the other hand, is specified relative to a point P by increments $\Delta_x$, $\Delta_y$, which are real; moreover, one of these increments is overscored, and is called the *cusp increments*; that is either $(\overline{\Delta}_x, \Delta_y)$ or $(\Delta_x, \overline{\Delta}_y)$ are valid cusp primitive notations. Geometrically, a cusp primitive is, as shown in Figure 2b, contained in a rectangle of dimensions $\Delta_x$, $\Delta_y$; the curved segment corresponding to the overscored increment is obtained by dividing the other increment into three equal parts, fitting a straight line in the middle section and a parabola in each of the other two sections so that the parabolas are tangent to the above straight line. More precisely, the cusp, $(\Delta_x, \overline{\Delta}_y)$, shown normalized in Figure 2c, is given, relative to a coordinate center at point P, by

In Region I $(0 \leq \frac{x}{\Delta_x} < \frac{1}{3})$;

$$\frac{y}{\Delta_y} = 1 - \left(1 - 3\frac{x}{\Delta_x}\right)^2 \qquad (a)$$

In Region II $(\frac{1}{3} \leq \frac{x}{\Delta_x} < \frac{2}{3})$;

$$\frac{y}{\Delta_y} = 1 \qquad (b)\ (2)$$

In Region III $(\frac{2}{3} \leq \frac{x}{\Delta_x} \leq 1)$;

$$\frac{y}{\Delta_y} = 1 - \left(3\frac{x}{\Delta_x} - 2\right)^2 \qquad (c)$$

The cusp $(\overline{\Delta}_x, \Delta_y)$ is obtained from Equations (2) by interchanging literal x with literal y everywhere in these equations. A cusp is always visible. These apparently mysterious primitives are justified on two counts: (i) ability to represent a large class of characters and symbols with a small number of primitives, as discussed immediately below and (ii) ease of implementation, as discussed in the following section.

A character or symbol is composed from a sequence of these two types of primitives; here the first primitive is specified relative to the lower left corner of the character field, and each subsequent primitive is specified relative to the terminating point of the preceding primitive. For example, capital letter A is formed in Figure 3a by the primitive sequence

$$S_A = (.45, 1.2)(.45, -1.2)\underline{(-.788, .3)}(.676, 0)$$

Observe that the first segment is a visible straight primitive which starts at the lower left corner and

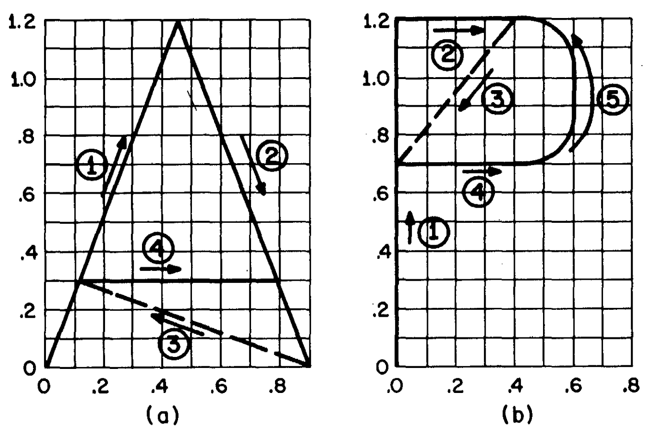


Figure 3—Character composition by straight/cusp primitives

terminates at the point [.45, 1.2]. The second segment is again a visible straight primitive, which starts as point [.45, 1.2] and terminates .45 units to the right and 1.2 units below that point. Observe further that the third segment is invisible, and that the direction and order in the sequence of each primitive is shown adjacent to each segment in Figure 3a. Capital letter P of Figure 3b is formed by the primitive sequence

$$S_P = (0, 1.2)\,(.4, 0)\,(\underline{-.4, -.5})\,(.4, 0)(.\bar{2}, 5)$$

Here, the first four primitives are straight with the third primitive invisible. The fifth primitive however is a cusp which starts at the point [.4, .7] and ends at the point [.4, 1.2].

Figure 4 shows the primitive sequences corresponding to all 94 alphanumeric characters and symbols of the ASC-II code. This Table is arranged exactly as the table of the ASC-II code for reference purposes. Some statistics of interest here are as follows:

1. The average number of primitive segments per character is 4.43.
2. The maximum number of primitives per character is eight.
3. The total number of different magnitudes for the primitive increments is 13.
4. No character uses more than two cusp primitives; these primitives occur (intentionally) either at the fifth, at the seventh, or at both the fifth and seventh segments of that character's primitive sequence.*

Of the above observations, 1, 2, and 3 indicate that a relatively small number of primitives can form a relatively large class of symbols. The fourth as well

* or they can be made to occur at these segment positions by introducing primitives (0, 0) anywhere in the sequence.

| | | | | | |
|---|---|---|---|---|---|
| | 0<br>$\underline{(.8,.9)}(-.8,.\bar{3})(0,-.5)(.8,-.\bar{4})(0,.5)$ | @<br>$\underline{(.5,.3)}(.3,.6)\underline{(0,-.6)}(-.3,.6)(-.\bar{3},-.6)\underline{(.1,-.3)}(-.\bar{6},1.2)(.2,-.3)$ | P<br>$(0,1.2)(.4,0)\underline{(-.4,-.5)}(.4,0)(.\bar{3},.5)$ | $\underline{(.5,1.0)}(-.2,.2)$ | p<br>$\underline{(0,-.3)}(0,1.2)\underline{(0,-.6)}(.8,-.\bar{3})(0,.3)(-.8,.\bar{3})$ |
| !<br>$\underline{(.4,1.2)}(0,-1.0)\underline{(0,-.1)}(0,-.1)$ | 1<br>$\underline{(.2,1.0)}(.2,.2)(0,-1.2)$ | A<br>$(.45,1.2)(.45,-1.2)\underline{(-.788,.3)}(.676,0)$ | Q<br>$\underline{(.8,0)}(-.2,.2)\underline{(.2,.7)}(-.8,.3)(0,-.5)(.8,-.\bar{4})(0,.5)$ | a<br>$\underline{(.8,0)}(0,.9)\underline{(0,-.6)}(-.8,-.\bar{3})(0,.3)(.8,.\bar{3})$ | q<br>$(.8,-.3)(0,1.2)(0,-.6)(-.8,-.\bar{3})(0,.3)(.8,.\bar{3})$ |
| "<br>$\underline{(.3,1.0)}(0,.2)\underline{(.2,-.2)}(0,.2)$ | 2<br>$\underline{(0,.9)}(.7,.\bar{3})(-.7,-.9)(.8,0)$ | B<br>$(0,1.2)(.4,0)\underline{(-.4,-.5)}(.4,0)(.\bar{3},.5)\underline{(0,-.5)}(.\bar{4},-.7)(-.4,0)$ | R<br>$(0,1.2)(.4,0)\underline{(-.4,-.5)}(.4,0)(.\bar{3},.5)\underline{(0,-.5)}(.4,-.7)$ | b<br>$(0,1.2)\underline{(0,-.6)}(.8,.\bar{3})(0,-.3)(-.8,-.\bar{3})$ | r<br>$\underline{(.1,0)}(0,.9)\underline{(0,-.2)}(.7,.\bar{2})$ |
| #<br>$\underline{(.15,0)}(.2,1.2)\underline{(.3,0)}(-.2,-1.2)\underline{(.25,.4)}(-.7,0)\underline{(.1,.4)}(.7,0)$ | 3<br>$\underline{(.1,1.2)}(.1,0)(.\bar{5},-.5)(.\bar{6},-.7)(-.2,0)$ | C<br>$\underline{(.8,.9)}(-.8,.\bar{3})(0,-.5)(.8,-.\bar{4})$ | S<br>$\underline{(.8,.9)}(-.8,.3)(.8,-.5)(-.8,-.\bar{4})$ | c<br>$\underline{(.8,.3)}(-.8,-.\bar{3})(0,.3)(.8,.\bar{3})$ | s<br>$\underline{(0,.3)}(.8,-.\bar{3})(-.8,.3)(.8,.\bar{3})$ |
| $<br>$\underline{(.4,1.4)}(0,-1.6)\underline{(.4,1.1)}(-.8,.\bar{3})(.8,.5)(-.8,-.\bar{4})$ | 4<br>$\underline{(.7,0)}(0,1.2)(-.7,-.9)(.8,0)$ | D<br>$(0,1.2)(.4,0)(.\bar{4},-1.2)(-.4,0)$ | T<br>$\underline{(.4,0)}(0,1.2)\underline{(-.4,0)}(.8,0)$ | d<br>$\underline{(.8,0)}(0,1.2)\underline{(0,-.9)}(-.8,-.\bar{3})(0,.3)(.8,.\bar{3})$ | t<br>$\underline{(.6,0)}(-.1,0)(-.1,.1)(0,1.1)\underline{(-.2,-.3)}(.4,0)$ |
| %<br>$\underline{(.1,0)}(.6,1.2)\underline{(-.6,0)}(0,-.2)\underline{(.6,-.8)}(0,-.2)$ | 5<br>$(.\bar{8},.7)(.1,.5)(.6,0)$ | E<br>$(0,1.2)(.7,0)\underline{(-.7,-.5)}(.6,0)\underline{(-.6,-.7)}(.8,0)$ | U<br>$\underline{(0,1.2)}(0,-.8)(.8,-.\bar{4})(0,.8)$ | e<br>$\underline{(0,.5)}(.8,0)(0,.1)(-.8,.\bar{3})(0,-.3)(.8,-.\bar{3})$ | u<br>$\underline{(.8,0)}(0,.9)\underline{(0,-.6)}(-.8,-.\bar{3})(0,.6)$ |
| &<br>$\underline{(.8,0)}(-.6,1)(.3,.\bar{2})(-.5,-.6)(.8,-.\bar{4})$ | 6<br>$\underline{(.6,1.2)}(-.2,0)(-.\bar{4},-1.2)(.\bar{4},.7)(-.4,-.2)$ | F<br>$(0,1.2)(.7,0)\underline{(-.7,-.5)}(.6,0)$ | V<br>$\underline{(0,1.2)}(.4,-1.2)(.4,1.2)$ | f<br>$\underline{(.3,0)}(0,1.0)\underline{(.2,-.1)}(-.3,0)\underline{(.1,.1)}(.4,.\bar{2})$ | v<br>$\underline{(0,.9)}(.4,-.9)(.4,.9)$ |
| '<br>$\underline{(.4,1.2)}(0,-0.2)$ | 7<br>$\underline{(0,1.2)}(.8,0)(-.6,-1.2)$ | G<br>$\underline{(.8,.9)}(-.8,.\bar{3})(0,-.5)(.8,-.\bar{4})(-.3,0)$ | W<br>$\underline{(0,1.2)}(.2,-1.2)(.25,.8)(.25,-.8)(.2,1.2)$ | g<br>$\underline{(.1,-.3)}(.5,0)(.2,.2)(0,.7)(-.8,.\bar{3})(0,-.3)(.8,-.\bar{3})$ | w<br>$\underline{(0,.9)}(.2,-.9)(.2,.9)(.2,-.9)(.2,.9)$ |
| (<br>$\underline{(.6,0)}(-.\bar{2},1.2)$ | 8<br>$\underline{(.8,.9)}(-.8,.\bar{3})(.8,-.5)(-.8,-.\bar{4})(.8,.5)$ | H<br>$(0,1.2)\underline{(0,-.5)}(.8,0)\underline{(0,.5)}(0,-1.2)$ | X<br>$\underline{(0,1.2)}(.8,-1.2)\underline{(-.8,0)}(.8,1.2)$ | h<br>$(0,1.2)\underline{(0,-.6)}(.8,.\bar{3})(0,-.6)$ | x<br>$(.7,.9)\underline{(-.6,0)}(.7,-.9)$ |
| )<br>$\underline{(.2,0)}(.\bar{2},1.2)$ | 9<br>$\underline{(.2,0)}(.2,0)(.\bar{4},1.2)(-.\bar{4},-.7)(.4,.2)$ | I<br>$\underline{(.2,0)}(.4,0)\underline{(-.2,0)}(0,1.2)\underline{(-.2,0)}(.4,0)$ | Y<br>$\underline{(0,1.2)}(.4,-.6)(.4,.6)\underline{(-.4,-.6)}(0,-.6)$ | i<br>$\underline{(.4,0)}(0,.9)\underline{(0,.3)}(0,0)$ | y<br>$\underline{(.1,-.3)}(.1,0)(.1,.1)(.5,1.1)\underline{(-.8,0)}(.4,-.9)$ |
| *<br>$\underline{(.2,.7)}(.4,.4)\underline{(.1,-.2)}(-.6,0)\underline{(.1,.2)}(.4,-.4)$ | :<br>$\underline{(.4,.6)}(0,0)(0,-.6)(0,0)$ | J<br>$\underline{(.3,1.2)}(.4,0)\underline{(-.2,0)}(0,-1.0)(-.4,-.\bar{2})$ | Z<br>$\underline{(0,1.2)}(.8,0)(-.8,-1.2)(.8,0)$ | j<br>$\underline{(.6,1.2)}(0,0)\underline{(0,-.3)}(0,-.9)(-.6,-.\bar{3})$ | z<br>$(.7,.9)(-.6,0)\underline{(-.1,-.9)}(.8,0)$ |
| +<br>$\underline{(0,.6)}(.8,0)\underline{(-.4,.4)}(0,-.8)$ | ;<br>$\underline{(.4,.6)}(0,0)\underline{(0,-.6)}(-.1,-.2)$ | K<br>$(0,1.2)\underline{(0,-.7)}(.7,.7)\underline{(-.5,-.5)}(.6,-.7)$ | [<br>$\underline{(.7,1.2)}(-.4,0)(0,-1.2)(.4,0)$ | k<br>$(0,1.2)\underline{(0,-.8)}(.8,.5)\underline{(-.6,-.4)}(.6,-.5)$ | {<br>$\underline{(.6,-.3)}(-.2,.2)(0,.5)(-.1,.1)(.1,.1)(0,.5)(.2,.2)$ |
| ,<br>$\underline{(.4,0)}(-.1,-.2)$ | <<br>$\underline{(.7,0)}(-.6,.6)(.6,.6)$ | L<br>$(0,1.2)\underline{(0,-1.2)}(.8,0)$ | \<br>$\underline{(0,1.2)}(.8,-1.2)$ | l<br>$\underline{(.4,0)}(0,1.2)$ | \|<br>$\underline{(.4,-.3)}(0,1.8)$ |
| -<br>$\underline{(0,.6)}(.8,0)$ | =<br>$\underline{(0,.8)}(.8,0)\underline{(0,-.4)}(-.8,0)$ | M<br>$(0,1.2)(.4,-.8)(.4,.8)(0,-1.2)$ | ]<br>$\underline{(.1,0)}(.4,0)(0,1.2)(-.4,0)$ | m<br>$(0,.9)\underline{(.4,-.3)}(0,-.6)\underline{(0,.6)}(-.4,.\bar{3})\underline{(.4,0)}(.4,.\bar{3})$ | }<br>$\underline{(.2,-.3)}(.2,.2)(0,.5)(.1,.1)(-.1,.1)(0,.5)(-.2,.2)$ |
| .<br>$\underline{(.4,0)}(0,0)$ | ><br>$\underline{(.1,0)}(.6,.6)(-.6,.6)$ | N<br>$(0,1.2)(.8,-1.2)(0,1.2)$ | ^<br>$\underline{(0,1.4)}(.4,.3)(.4,-.3)$ | n<br>$(0,.9)\underline{(0,-.3)}(.8,.\bar{3})(0,-.6)$ | ¬<br>$\underline{(0,.2)}(.8,0)(-.2,0)$ |
| /<br>$(.8,1.2)$ | ?<br>$\underline{(.4,0)}(0,.1)\underline{(0,.1)}(0,.2)(.3,.5)(-.8,.\bar{3})$ | O<br>$\underline{(.8,.9)}(-.8,.\bar{3})(0,-.5)(.8,-.\bar{4})(0,.5)$ | _<br>$\underline{(0,-.2)}(.8,0)$ | o<br>$\underline{(0,.3)}(0,.3)(.8,.\bar{3})(0,-.3)(-.8,-.\bar{3})$ | |

Figure 4—Straight/cusp primitive sequences for 94-character ASC-II set

as the other observations above will be used in the following section in connection with the implementation of this character generation technique.

### *The character generator*

A local character generator for a CRT display is generally a system (Figure 5) with input a seven-bit word, denoting a character, and output two deflection and one beam-intensification waveforms (functions of time), which when applied to the CRT deflection and beam controls, respectively, display that character relative to beam position, $x_p$ and $y_p$. Character and line spacing is usually accomplished by a control unit external to the generator, which varies $x_p$ and $y_p$ upon completion of each character and line, respectively. If the CRT display module is of the refresh type, then the codes of characters to be displayed are stored in a local storage medium, usually a delay line, and are presented periodically, usually every 1/30 to 1/40 sec to the character generator. If the CRT display module is of the storage type, then the character generator generates the waveforms x, y and b only once for each character to be displayed, and the corresponding character is stored on the screen of the CRT.

Any given character primitive $y = f(x)$ can be generated by such a system in an infinite number of ways, since for every one of many possible choices for a horizontal deflection waveform x(t), where t is time, there is always a vertical deflection waveform $y(t) = f(x(t))$ which when applied simultaneously with x(t), causes the CRT beam to trace the primitive $y = f(x)$. Two particular types of waveforms, s(t) and c(t) were chosen to implement the primitives of the preceding section; they are shown in Figure 6a, and their time derivatives in Figure 6b.

A straight-line primitive about any point is generated by applying waveform s(t), appropriately scaled, to both the horizontal and vertical axes. Thus, setting

$$x(t) = \Delta x\, s(t) + x_1 \quad \text{(a)}$$
$$y(t) = \Delta y\, s(t) + y_1 \quad \text{(b)} \qquad (3)$$

where Δx and Δy are real numbers, results in a straight line primitive from $[x_1, y_1]$ to $[x_1 + \Delta x, y_1 + \Delta y]$ given by

$$\frac{x - x_1}{\Delta x} = \frac{y - y_1}{\Delta y} \qquad (4)$$

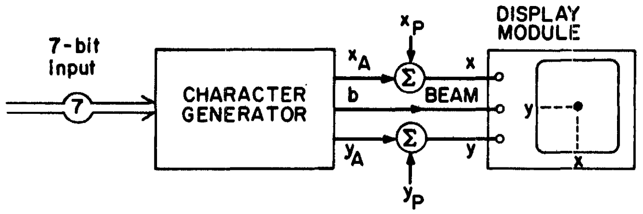


Figure 5—Local character generator

and shown in Figure 6c. This is the desired primitive of Equation (1).

A cusp primitive about any point, is generated by applying waveform s(t) to one axis and waveform c(t) to the other, after each waveform has been appropriately scaled. Figure 6d shows the resulting segment when s(t) is applied to the horizontal axis, and c(t) to the vertical axis, and Figure 6e shows a segment obtained with different scaling and interchange of the two waveforms. More generally, setting

$$x(t) = \Delta x s(t) + x_1 \quad \text{(a)}$$
$$y(t) = \Delta y c(t) + y_1 \quad \text{(b)} \qquad (5)$$

where Δx and Δy are real numbers, yields a cusp primitive, about-point $[x_1, y_1]$ described as follows:

for

$$0 \leq \frac{x - x_1}{\Delta x} < \frac{1}{3},$$

$$y = y_1 + \Delta y \left[1 - \left(1 - 3\frac{x - x_1}{\Delta x}\right)^2\right] \quad \text{(a)}$$

for

$$\frac{1}{3} \leq \frac{x - x_1}{\Delta x} < \frac{2}{3}, \qquad y = y_1 + \Delta y \qquad \text{(b)} \quad (6)$$

for

$$\frac{2}{3} \leq \frac{x - x_1}{\Delta x} \leq 1,$$

$$y = y_1 + \Delta y \left[1 - \left(3\frac{x - x_1}{\Delta x} - 2\right)^2\right] \quad \text{(c)}$$

Equation (6) is identical in form to the desired cusp primitive, given by Equation (2). Since Equations (4) and (6) implement exactly all the primitives of the previous section, about any point $(x_1, y_1)$, it remains only to provide means for forming a string of

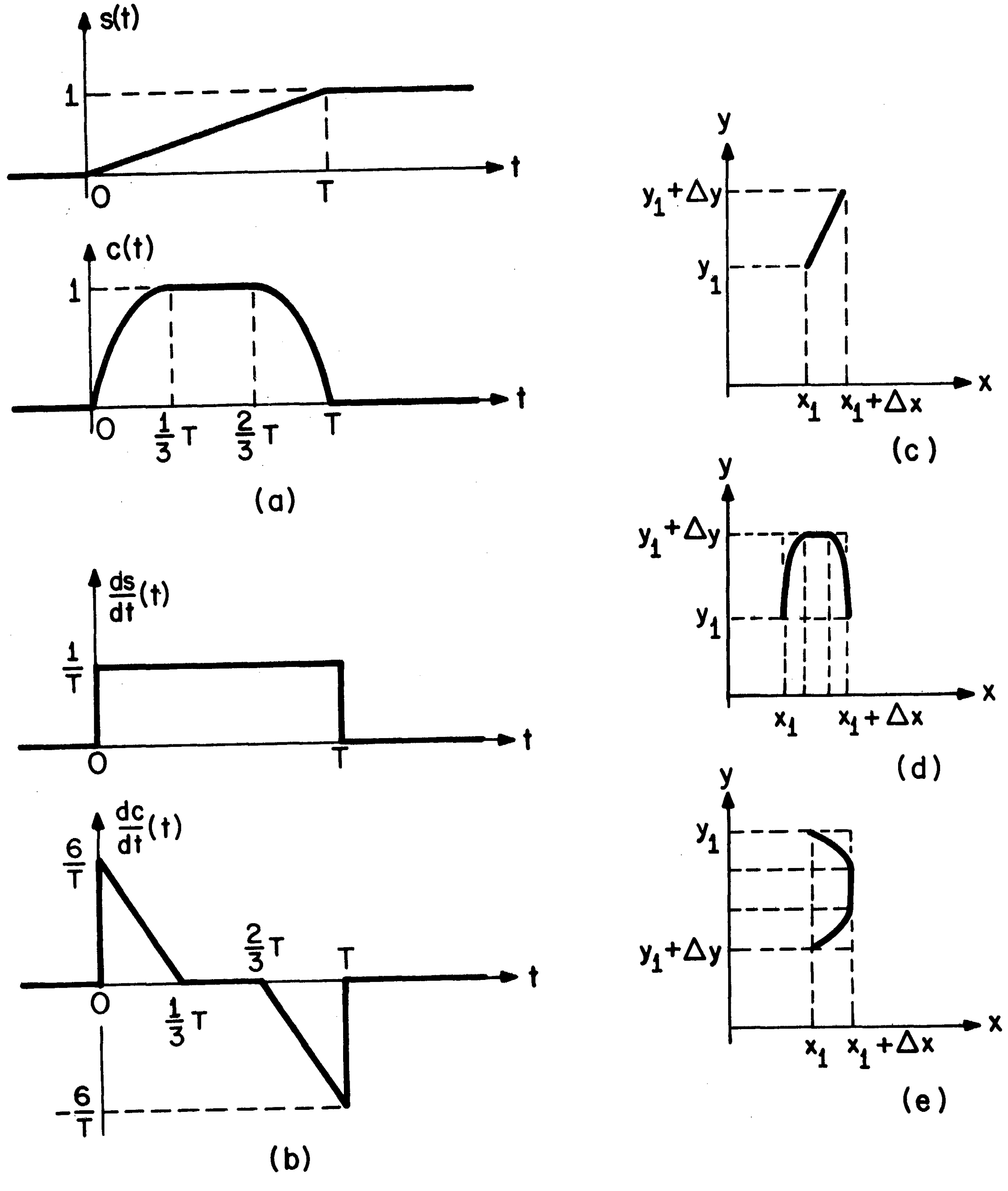


Figure 6—Waveforms for straight/cusp primitives

primitives, so that all the characters of Figure 4 may be implemented.

The formation of strings of primitives, that is of characters and symbols, is accomplished by concatenating the derivative waveforms of Figure 6b, for each primitive segment, after they have been scaled by $\Delta x$ and $\Delta y$. Such waveforms, denoted by $(1/T)dx/dt$ and $(1/T)dy/dt$, (T constant) are shown for letter P

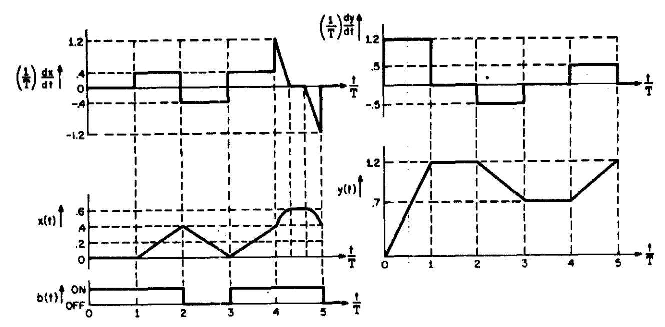


Figure 7—Composition of CRT deflection and beam waveforms

on the top half of Figure 7; subsequent integration in time of these waveforms yields the deflection waveforms x(t) and y(t), shown on the lower half of Figure 7. Also shown in Figure 7 is the beam waveform b(t) which turns the beam off in the third time segment $2T \leq t \leq 3T$. The character resulting from simultaneous application of these x(t) and y(t) waveforms on the CRT is the letter P of Figure 3b, specified by the primitive string:

$$S_P = (0, 1.2)(.4, 0)(\underline{-.4, -.5})(.4, 0)(.\bar{2}, .5)$$

Observe that these five primitives correspond to and are ordered as the five time segments of Figure 7.

One way of implementing this character-generation approach is shown in Figure 8. Here, sixteen lines carry eight rectangular constant-amplitude voltage pulses $P_i$, and their negatives and four lines carry two cusp-derivative pulses $C_i$ and their negatives. Waveforms and relative timing of these pulses are shown on the top center of Figure 8. Operation of the system is as follows: a character to be displayed is specified to the decoder shown on the right side of Figure 8, by, a seven-bit binary word. This word is "decoded", so that one of the 128 output lines of the decoder, say the line marked P, becomes energized. That line, turns "on" the three analog switching devices to which it is connected, and starts the timing sequences of the $P_i$ and $C_i$ pulses. The dx/dt, dy/dt and b waveforms for the selected character are formed by resistive mixing of the above pulses in three groups, respectively. For the case under discussion, letter P is "stored" in the values and manner of interconnection of eight resistors shown enclosed by dashed lines. Here, the top four resistors mix pulses $P_2$, $-P_3$ and $P_4$, all equally weighted by a conductance of .4 units; the fourth resistor in that group weighs waveform $C_5$ by .2. As a consequence of this mixing, the resulting current in the so-called xbus is the weighted sum of all these waveforms and is identical to the dx/dt waveform of Figure 7. The next group of three resistors having conductances 1.2, .5 and .5 respectively forms, in a similar manner a current in the ybus which is the dy/dt waveform of Figure 7. Finally, the complement of the beam waveform of Figure 7 is formed by the last group consisting of one resistor of unit conductance, as a current in the b bus. The dx/dt and dy/dt currents are subsequently amplified by low-input-impedance amplifiers A and integrated in time to yield the $x_A(t)$ and $y_A(t)$ waveforms of Figure 8. These are identical to the desired x(t) and y(t) waveforms of Figure 7. These waveforms are, in turn, summed with the constants $x_P$, $y_P$ and the beam waveform is inverted resulting in a display of character P about point $[x_P, y_P]$. At the end of this sequence, the integrators are reset to zero output and the analog switching devices are turned off, thereby making the character generator ready for display of the next requested symbol. Also shown in Figure 8 is the resistor "memory" for character 1; the reader may verify that when this character is selected, the system does indeed generate the primitive sequence for that character, shown in Figure 4. Observe finally that the system of Figure 8 has two rather than eight cusp lines, $C_i$, which are active at timing positions five and seven. The reason for this choice is one of economics, since as we discussed in connection with Figure 4 it has been established over a large class of characters and symbols that these pulses at such relative positions are quite adequate.

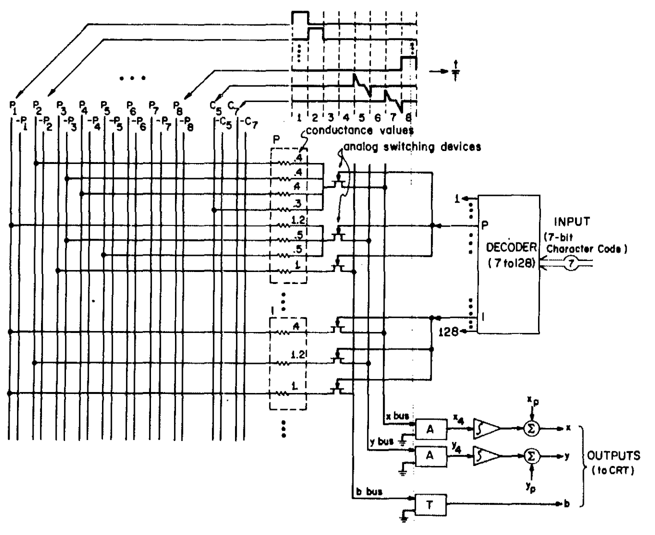
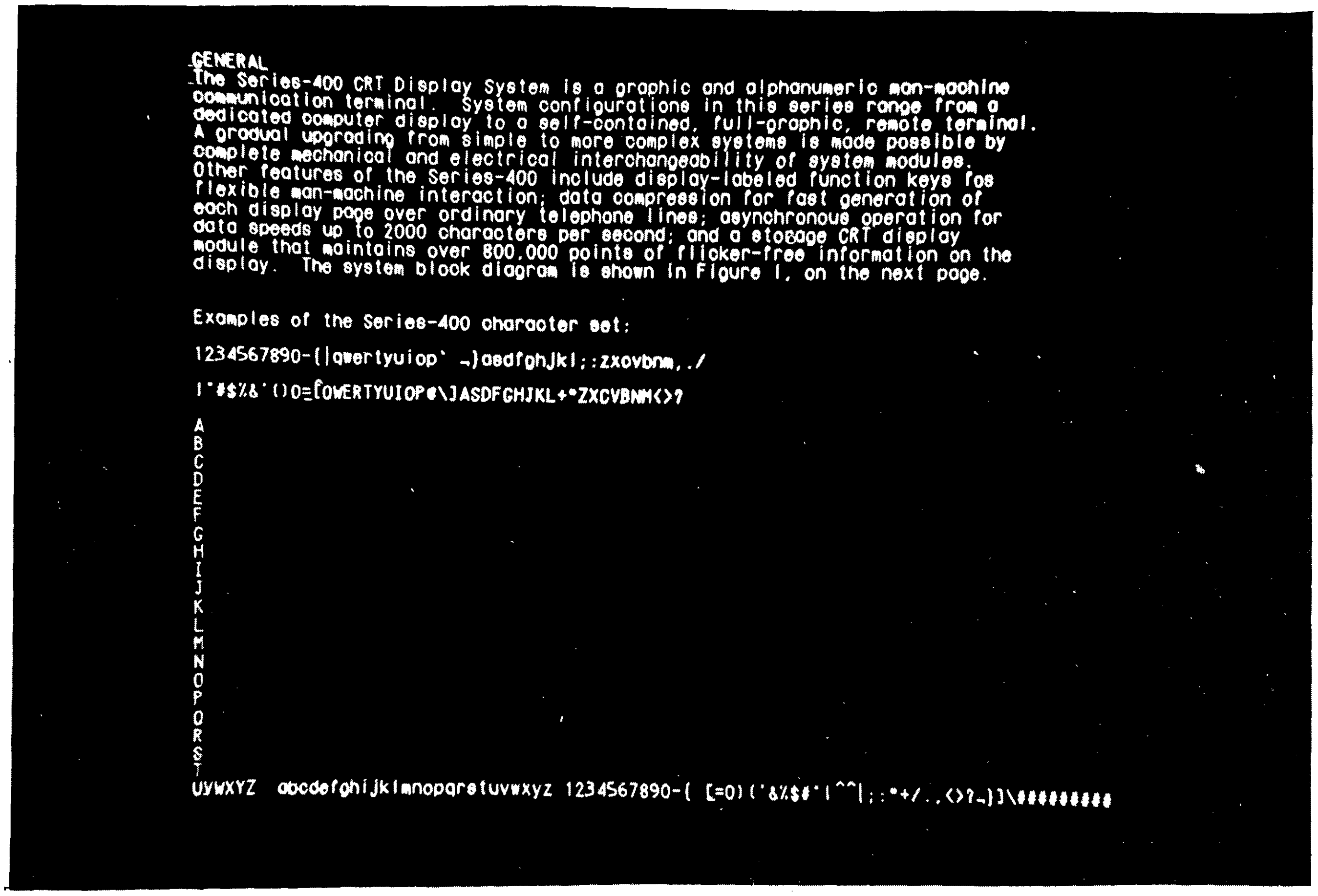


Figure 8—Character generator implementation



```
GENERAL
The Series-400 CRT Display System is a graphic and alphanumeric man-machine
communication terminal.  System configurations in this series range from a
dedicated computer display to a self-contained, full-graphic, remote terminal.
A gradual upgrading from simple to more complex systems is made possible by
complete mechanical and electrical interchangeability of system modules.
Other features of the Series-400 include display-labeled function keys fos
flexible man-machine interaction; data compression for fast generation of
each display page over ordinary telephone lines; asynchronous operation for
data speeds up to 2000 characters per second; and a stoBage CRT display
module that maintains over 800,000 points of flicker-free information on the
display.  The system block diagram is shown in Figure 1, on the next page.

Examples of the Series-400 character set:

1234567890-(|qwertyuiop` ¬)asdfghjkl;:zxcvbnm,./

!"#$%&'()0=[QWERTYUIOP@\]ASDFGHJKL+*ZXCVBNM<>?

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
UVWXYZ  abcdefghijklmnopqrstuvwxyz 1234567890-( [=0)('&%$#'|^^|;:*+/.,<>?¬]]\#########
```

Figure 9—Implemented characters (Courtesy of Computek Inc., Cambridge, Mass.)

A photograph of characters and symbols generated by such a system is shown in Figure 9.

An alternative realization of the above character-generation technique would be to store for each character k bits in a digital read-only memory. These bits would, in turn, control a common, over all characters, resistive mixing network, by varying in discrete steps the conductances of this network. Such an implementation, however, requires approximately $k = 90$ bits of storage per character and is considerably less economical than the system of Figure 8.

## COMPARISONS AND CONCLUSIONS

Ultimately, the merits and disadvantages of a character generator rest on economic and aesthetic criteria. The former are very strongly dependent on technology and are subject to rapid change, while the latter are, beyond a certain point, quite subjective. Nevertheless, certain conclusions can be drawn.

First, the use of stroke primitives such as straight lines and cusps results in more economical character storage than the use of points; and the relative advantage of such storage increases, over a certain range, with finer resolution. Consider for example that every character is formed on a grid of $n^2$ points. A straightforward point-intensification or incremental-stroke scheme on such a "dot-matrix" would require the

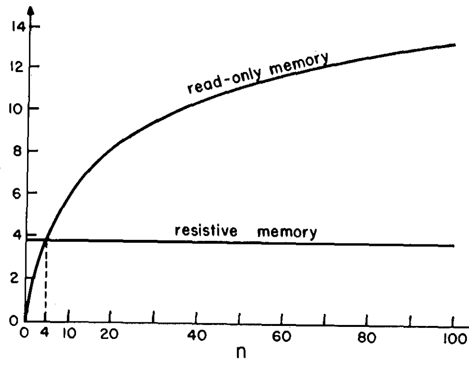


Figure 10—Memory growth versus resolution

storage of $n^2$ bits per character, indicating the points "which must be intensified—the corresponding memory growth curve, giving the number of diode components per character, is shown in Figure 10 and is labeled read-only memory". From Figure 4, however, we know that the average number of segments per character, over the 94 character ASC-II set, is 4.43 for the approach of this paper. Each segment, in turn requires two resistors, for x and y. We also know from Figure 4 that there will be of the order of 1.4 resistors per character for the beam. Hence, the average number of resistors per character is constant or

$$2(4.4) + 1.4 \cong 10$$

In addition, each character requires three analog switching devices (FETS) and their driver, or the order of five components. Thus, the total number of components is 15 per character, remaining constant within the limit of analog resolution, or $n \leq 100$, as shown by the graph labeled "resistive memory" in Figure 10. With present technology, it is more economical to construct the character generator out of discrete components; the resulting cost, is for an acceptable resolution $n^2 = 240$, lower than that of a read-only memory of ¼ the resolution. With forthcoming technology, the above ten resistors and five active components, should cost each about as much as a diode, hence an even better cost advantage can be expected. Observe however, as was indicated above, that resolution cannot exceed that of analog circuitry, since the storage and generation of characters is analog in nature. On the other hand, the CRT is an analog device, on which resolutions higher than analog cannot be effectively used. The above savings in character storage, result in lower generator cost, and reduced generator size.

Second, the speed of character generation of such a stroke technique is of the same order as that of dot-intensified character generation, since the current through resistors will generally change over its minimum resolvable increment as rapidly as, or faster, than the full current swing through a diode.

Third, the mixing of *time-derivative* waveforms, and the subsequent integration of these waveforms provides good character appearance through suppression of spurious noise and continuity of the integrated waveforms.

Finally, the fidelity of continuous-stroke characters with the above primitives is considerably higher than that of dot-intensified, or incremental-vector characters of comparable resolution. Such a comparison can be made visually by the reader on Figure 11 for a resolution of $n = 4$, or a grid of 16 points.

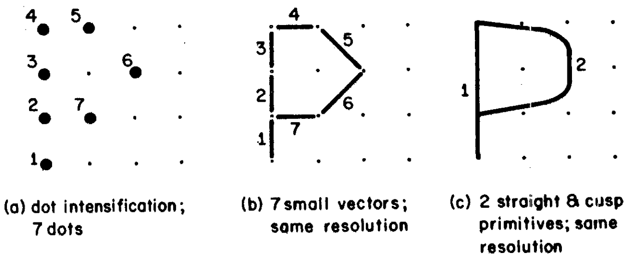

4
5
3
6
2
7
1
(a) dot intensification; 7 dots
4
5
3
2
6
1
7
(b) 7 small vectors; same resolution
1
2
(c) 2 straight & cusp primitives; same resolution

Figure 11—Comparison of straight/cusp and point intensified characters of same resolution

The approach discussed in this paper can be further extended to a more complete hardware "grammatical" structure, through a straightforward extension. That is, characters can be constructed from primitives and other simpler constructs which are themselves composed of primitives and/or other constructs of the same class. For example, as seen from Figure 4, the primitive sequence, $S_8$, for numeral 8 contains the primitive sequence $S_S$ for capital S. That is, $S_8 = S_S$ (.8, .5) It is not yet clear whether such a hardware structure will result in even lower cost, without sacrifice of performance.

Finally, we would like to close with the philosophical observation that the use of sizable straight-line and cusp primitives is well suited to character generation, since characters and symbols were generated, on the first place, through such strokes, by pen or stick, on paper or sand, rather than by dots or by infinitesimal straight-line segments.

## ACKNOWLEDGMENTS

I wish to thank Dr. H. L. Graham of Computek Inc., for his contributions to the design and construction of a prototype generator; Mr. Ben Simpson of Computek Inc., for the design and development of a practical generator, and Mr. L. Dounias of Ashely Meyer & Associates for his aesthetic design of the character set. This work was performed on a consulting basis for Computek Inc., 143 Albany Street, Cambridge, Massachusetts.

# Economical display generation of a large character set

*by* KOJI NEZU and SACHIO NAITO

*Nippon Electric Company, Ltd.*
Kawasaki, Japan

## INTRODUCTION

Electronic computers find wide applications in the fields such as document production, compilation of printed articles, language translation. The need for high speed printers and display systems for various types of characters and symbols is increasing. There is a high demand for high speed printers and display systems for "Kanji (Chinese characters)" in Japan and other Oriental countries. A character generator with a font capacity greater than 1000 is required. Nearly the same number of character fonts might be needed also in Western countries, if special fonts of Greek or Roman alphabets, italics, bold face, or special mathematical symbols are included.

A pattern generator is of prime importance because it carries out the translation from binary coded information into characteristic and symbolic information. The pattern generator is important as it is an initial link. It is contained in a printer and acts to connect the computer and printer. It is of paramount importance that the pattern generator has high speed and high resolution.

High speed pattern generation systems with the font capacity greater than 1000 have been reported. However, they were designed for typesetting and consequently too expensive.[1,2] In a typical system a flying spot scanner and a character grid are utilized. The character grid is a pattern carrying film or glass plate on which a number of character patterns are printed in a negative form. The flying spot scanner must exhibit a very high resolution in order to discriminate more than 1000 characters and symbols stored in the character grid. The electronic circuits must be of high quality in order to achieve precise control of scanning. This makes high speed pattern generation systems expensive and bulky. In this paper we would like to present a new pattern generating system which is low in cost and compact in size.

### *System concept*

The attempt to decrease the cost and size of pattern generators was carried out by replacing the expensive flying spot scanner and associated circuits with other means. One such means would be a vidicon, a photoelectric conversion device, in which a light-irradiated photoconductive element is scanned by an electron beam producing a video signal. However, the vidicon has no resolution which can be attained with a flying spot scanner. It cannot discriminate more than 1,000 characters and symbols when they are projected simultaneously on the face of the vidicon.

It is apparent that the resolution and discrimination characteristics can be improved if a small number of larger patterns are projected on the face of the vidicon. The use of larger patterns keeping the total number of characters constant, is achieved by introducing character grids. Each grid shares the total number of characters and symbols, and the grid is selected and projected on the face of the vidicon by a flash tube selectively energized.

Vidicons retain residual image. This property is unfavorable for the time-shared use of several character grids. Thus the residual image formed on the face of the vidicon by a prior projection should be erased

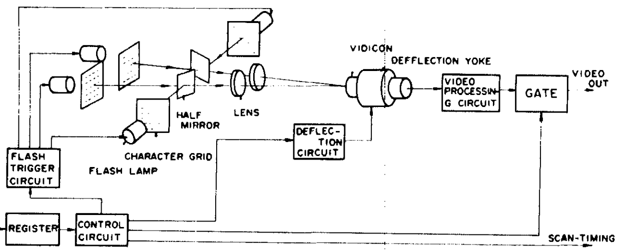


Figure 1—Schematic diagram of the character generator

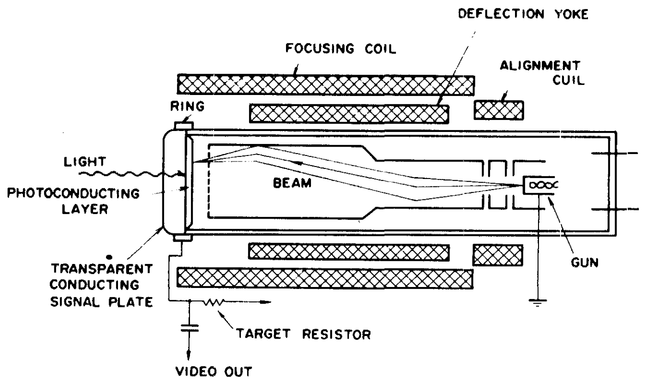


Figure 2—Schematic cross-section of the vidicon

before the succeeding projection of a different character grid is made. Accordingly, the erasing scan of the vidicon face should precede each reading scan. However, if an erasing scan is applied to the entire face of the vidicon, a certain time would be wasted prior to each reading scan. In our pattern generating system the erasing scan or prescanning is restricted to the area where the reading scan is to follow. The remaining area is occupied with residual image.

*Character generating unit*

Figure 1 shows a schematic diagram of the character generator designed for Kanji. Characters and symbols are printed in a 16 by 16 matrix form on each of four character grids. Four miniature flash lamps whose light emission timing is determined by a control circuit are used to project the real image of the four character grids. When one of the flash lamps is selectively energized, all the patterns printed on the corresponding character grid are projected on the full effective area of a target face of the vidicon (type 8572) by means of a half mirror and a lens having reduction ratio 1/2. F-number of the lens is 5.6.

*Generating cycle*

The vidicon consists of a highly evacuated envelope containing an electron gun at one end and a transparent optical flat target face at the other (Figure 2). A transparent conductive layer is deposited on the inner surface of the target face as a signal plate. A photoconductive film is deposited on this layer so as to form capacitors. In the site of electron impact the surface of photoconductive layer catches a negative charge of electron. When no light falls on the photoconductive layer, its surface is maintained at the cathode (ground) potential by electron beam scanning because the layer is a good insulator. When a pattern is projected conduction increases in the bright areas. The bright part of the pattern enhances the leakage current through the layer and let the capacitors discharge during exposure. The reading scan which follows the exposure restores the negative charge, and the current for the restoration produces a video signal across the target resistor.

Generation of a character is accomplished by the following sequential steps:

1. Prescanning the area where the desired character is to be projected.
2. Flashing a xenon lamp in order to project and store the character image on the vidicon target.
3. Scanning the area of a particular character in order to pick up the video signal.

In the prescanning step, the deflection yoke moves the electron beam to the position on the vidicon target where the character is to be projected, and lets the beam form a small raster throughout the area to cover the image of character. The raster size is 0.7 mm square (about 1/250 of full effective area of the vidicon target). It takes 1.5 ms to erase completely the residual image stored by preceding flashes.

Two factors are specified for the image persistence, viz., the transient response of photoconductive material, and the time lag which results from incomplete charging of electrons on the target with large capacitance by the scanning beam of low landing efficiency.[3,4] Generally the photoconductive decay time constant is very short, of the order of one ms. On the other hand, the capacitive lag makes a predominant contribution to the image persistence (of 10 ms) in the standard TV application. Since the total target capacitance is proportional to the size of the raster, there is no significant capacitance in the present application where about 1/250 of the total surface is used. The localized scanning reduces the resultant image persistence time from 10 ms of TV application to about one millisecond.

The flashing illumination just after the prescanning continues only 5$\mu$s. Each miniature xenon-flash lamp is energized by the discharge of a capacitor, which is triggered by a selection pulse. Although more than two hundred characters are projected on the target face of a vidicon, only one of them is exactly stored on the vidicon target, because the corresponding part of the vidicon target has been presecanned.

The last step is the reading scan. The deflection of the scanning beam during this step is the same as that of the prescanning. However, the videosignal on the output of the vidicon, is taken out through a video gate circuit.

Figure 3 illustrates a portion of a real image. It is projected from a character grid onto the target face of the vidicon. Owing to the image persistence of the vidicon target, the image focused persists for a certain period even if the projection is executed for a very short time. The prescanning and read scanning of particular area are accomplished by the X and Y deflection yoke. The prescanning and read scanning modes are illustrated in Figure 3. Examples of prescanning and read scanning are shown by the lines superimposed on the letter 3.

The deflection of the scanning beam to any position on the vidicon target can be accomplished in 5$\mu$s. Linearity and stability of the deflection amplifier are approximately 0.1 percent. The bandwidth of deflection amplifier is 500 KHz.

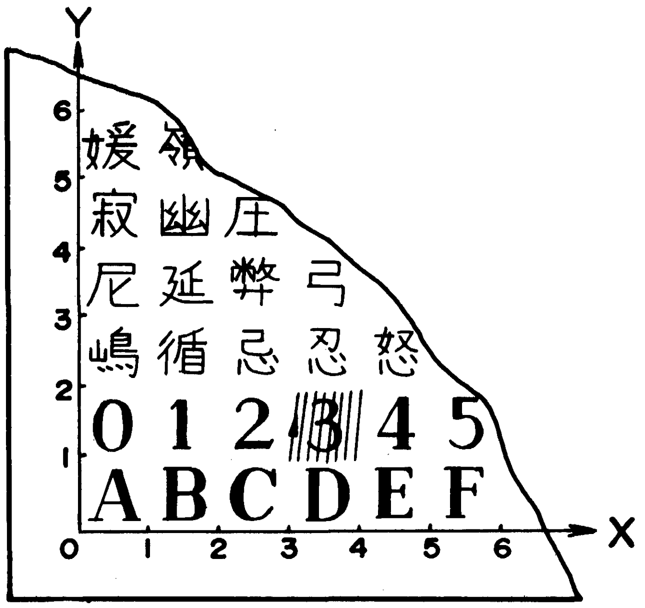


Figure 3—Figure of the image of a character grid projected on the face of the vidicon

Perpendicularity and residual magnetism of the deflection yoke, and pin cushion or barrel distortion of the vidicon are the other factors influencing the positioning accuracy of the electron-beam deflection. The pin cushion distortion of the vidicon diminishes the accuracy considerably. Four small magnets each of the size 2 × 2 × 3 mm placed close to the vidicon target correct the distortion. Residual magnetism of each magnet is about 2000 G along the longitudinal axis. Positions of magnets are adjusted by means of screws. In the present system the overall error of beam positioning is kept within 0.5 percent of full deflection. This is sufficient because the projected characters are larger than those in the flying spot system.

*Processing of video signal*

The video signal output of the vidicon is amplified and converted to a two-level signal by a video-processing circuit.

As the aperture of the scanning electron beam is not very sharp, the video signal contains an intermediate level notwithstanding the fact that the character grid has two levels of black and white. Considerable variations in both modulation depth and dc level occur in the video signal depending upon detailed patterns of projected characters. Shading of the vidicon also causes variations. Thus a simple clipping circuit of constant clipping level cannot be used.

Figure 4 shows a block diagram of the video processing circuit. The increment of the video signal is detected in the differentiation circuit which consists of a 0.5 $\mu$s delayline and an integrated differential amplifier $\mu$PC53. This circuit eliminates the dc-level shift from the video signal and sends trigger pulses to a flip-flop which converts the video signal to a two-level signal.

## System operation

The control circuit in Figure 1 decodes a pattern-representing binary signal, selectively energizes one of the flash lamps, and controls the prescanning and read scanning in the vidicon so that the desired one of the projected patterns is scanned.

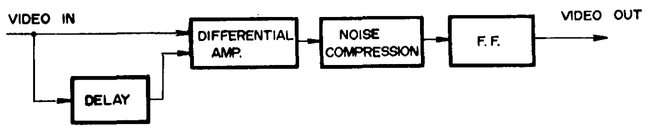


Figure 4—Block diagram of the video processing circuit

Two significant bits of the character-representing signal are decoded into four flash lamps to select one character grid out of four character grids. The remaining eight bits are supplied to the X and Y direction D-A converters in the deflection circuits.

There are two saw-tooth waveform generators in the control circuit, one for X-scanning and the other for Y-scanning. The repetition frequency of Y-scanning is 20 KHz while that of X-scanning is 0.67 KHz. The ratio of these frequencies is determined by the number of scanning lines for one character. In the present system, each character is scanned by 30 vertical lines. The scanning signals from each saw-tooth wave generators are respectively added to the character selection signals of X- and Y-axis which are supplied from the D-A converters. Figure 5 shows a block diagram of the deflection circuit.

The control circuit produces the gate pulse for the video signal as soon as the read-scanning starts. Synchronizing pulses for X- and Y-axes are available from the control circuit for the reconstruction of the character images either at display or at printer unit.

*Operating characteristics.*

The optical unit of character generator is shown in Figure 6. The size of this unit is 500 mm wide, 600 mm long and 150 mm high. The weight is 20 kilogram. Almost all the electronic circuits are constructed by IC's.

The quality of the characters generated by the present 1024 font capacity system is sufficiently high. Figure 7 shows an example of Japanese sentences displayed on a CRT display unit. The storage CRT which needs no costly memory devices for the refreshment of the information, is suitable for this application. Figure 8 shows an example of printed pages performed by a fiber optics CRT unit.

The generating speed of the present system is 330 characters per second. The machine speed is restricted by the persistent lag in the vidicon. In order to decrease the time for erasing a new photoconductive layer of the vidicon is required.

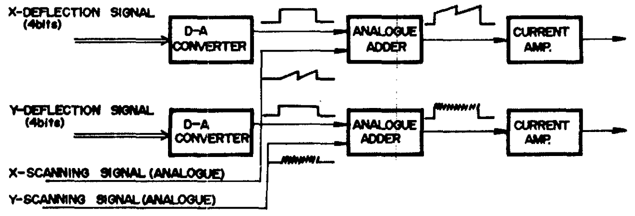

X-DEFLECTION SIGNAL (4bits)
D-A CONVERTER
ANALOGUE ADDER
CURRENT AMP.
Y-DEFLECTION SIGNAL (4bits)
D-A CONVERTER
ANALOGUE ADDER
CURRENT AMP.
X-SCANNING SIGNAL (ANALOGUE)
Y-SCANNING SIGNAL (ANALOGUE)

Figure 5—Block diagram of the deflection circuit

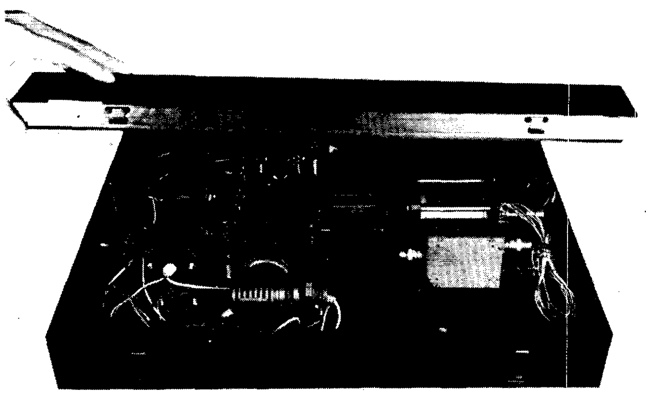
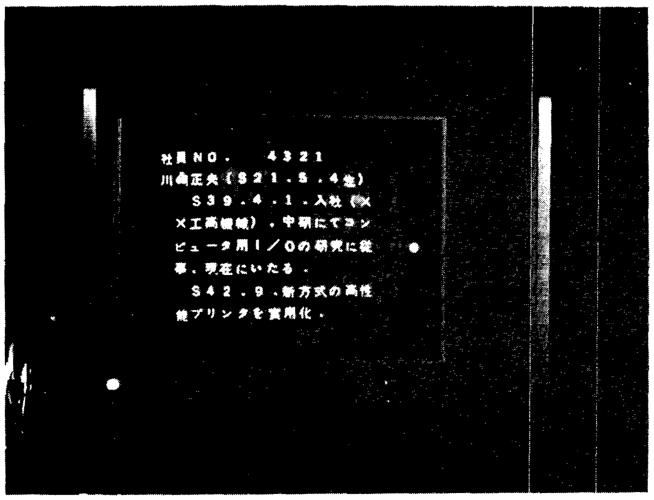


Figure 6—The optical unit of character generator



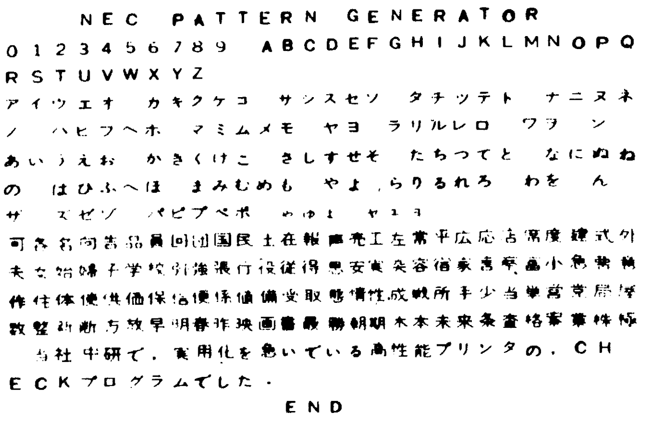
社員NO. 4321
川崎正夫(S21.5.4生)
S39.4.1.入社([illegible]
X工高機械). 中研にてコン
ピュータ用I/Oの研究に従
事. 現在にいたる.
S42.9.新方式の高性
能プリンタを実用化.

Figure 7—Displayed Japanese sentences on CRT

The reliability of the vidicon operated under unusual condition of selective scanning on the target was investigated by the running test of about 1000 hours. But no noticeable change was observed.

## CONCLUSION

It has been confirmed that the new opto-electronic character generating system with 1024 font capacity has many advantages such as high font capacity, high speed, high quality, low cost and small size. The advantages have been achieved by utilizing four character grids and one single vidicon. Each character grid



```
      N E C   P A T T E R N   G E N E R A T O R
0 1 2 3 4 5 6 7 8 9   A B C D E F G H I J K L M N O P Q
R S T U V W X Y Z
ア イ ウ エ オ  カ キ ク ケ コ  サ シ ス セ ソ  タ チ ツ テ ト  ナ ニ ヌ ネ
ノ  ハ ヒ フ ヘ ホ  マ ミ ム メ モ  ヤ ヨ  ラ リ ル レ ロ  ワ ヲ  ン
あ い う え お  か き く け こ  さ し す せ そ  た ち つ て と  な に ぬ ね
の  は ひ ふ へ ほ  ま み む め も  や よ ，ら り る れ ろ  わ を  ん
ザ  ズ ゼ ゾ  パ ピ プ ペ ポ  ゃ ゅ ょ  ャ ュ ョ
可各名向書品員回団国民土在報画発工左常平広応占常度建式外
夫女始婦子学校引施張行役従得思安実発容宿家寿寒晶小急帯輸
作住体使供価保治便係値備受取態情性成戦所手少当単高京層学
数整新断方放早明春作映画書最勝朝期木本未来条査格審業株極
  当社中研で，実用化を急いでいる高性能プリンタの，C H
E C K プ ロ グ ラ ム で し た ．
                     E N D
```

Figure 8—An example of printed pages performed by a fiber optics CRT unit

contains 256 characters, and one grid is selectively projected on the vidicon by the combination of half-mirrors and flash lamps. A vidicon is used instead of an expensive flying-spot scanner to convert projected characters into video signals.

The problem of image persistence in the vidicon was solved by a localized prescanning which quickly makes the target ready for projection.

The generation speed of 330 characters per second was realized with the new character generating system. Excellent stability was confirmed for a long period of operation.

## ACKNOWLEDGMENT

The authors wish to thank Dr. Shigeru Sekiguchi and Mr. Yasukuni Kotaka for their kind interest in this work, and Mr. Tomoyuki Watanabe for his cooperation in the circuit design of the device.

## REFERENCES

1 T TAKAHASHI J HASEGAWA
*A design of high speed Kanji printer*
Proc 33rd FID Conf and International Congress on Documentation Sept 1967

2 G D FRIEDLANDER
*Automation comes to the printing and publishing industry*
IEEE Spectrum 48-62 April 1968

3 Y KIUCHI
*The persistent lag in the vidicon*
Toshiba Review Vol 13 920-926 1958

4 R W REDINGTON
*The transient response of photoconductive camera tubes employing low-velocity scanning*
IRE Trans ED4 July 220-225 1957

# ISDS—A program that designs computer instruction sets

*by* F. M. HANEY

*Scientific Data Systems*
El Segundo, California

## INTRODUCTION

ISDS (Instruction Set Design System), a program that designs instruction languages for computers, is the result of research aimed at gaining a better understanding of computer-assisted design and, in particular, automated design of computers. The primary goal of the research was to develop techniques for writing programs that solve design problems without intervention by human designers. This paper describes a program that solves a specific design problem—the selection of an order code for a computer—but the general approach can be easily adapted to other design problems.

ISDS contains a generalized model of a computer instruction set and solves a design problem by filling in details of the model, analyzing the result with respect to the requirements of the given problem, and selecting instances of the model that best meet the requirements of the problem.

The model used by ISDS is GIS (Generalized Instruction Set) which is capable of representing a broad range of computer instruction sets, including most of the features of existing computers.

The programs that make up ISDS operate at several levels, the lowest of which is used to manipulate the tree structures storing GIS representations of instruction sets. Other programs in ISDS perform computations useful in analysis of instruction sets, select values for single values of the instruction set, analyze an entire instruction set, and determine the optimal method of selecting parts of the instruction set.

This paper is organized into four sections:

1. A discussion of design theory and the basis for the ISDS approach to constructing design programs.
2. A description of GIS, the model of an instruction set that ISDS uses as its basic design concept in solving a problem.
3. A description of the programs that make up ISDS and the actual operation of ISDS, including an example of an instruction set designed by ISDS.
4. A summary of the results of experimentation with ISDS.

### *Formalizing the design process*

Before programs that simulate design process can be considered, the complex nature of this process must be understood. Many models of the design process have been proposed, but for the most part they are the same in content if not in detail. However, two men have adequately expressed the complexity of design process—Asimow[1] and Alexander.[2]

Asimow considers design as a process of specification during which the solution to a design problem is gradually transformed from an abstraction into a physical reality. At each step the solution is analyzed. If any part of the solution fails to meet some requirement of the design problem, or if other decisions lead to better solutions, some parts of the solution may have to be re-specified. Dealing primarily with engineering design, Asimow identifies over 25 different steps in the design process, each dealing with a different level of detail.

Alexander's view of design is consistent with Asimow's, although Alexander places greater emphasis on the relationships between design variables which must be considered at each step. The value of any part of a solution depends on, and may help to determine, the value of other parts of the solution. Since design is generally a serial process, the designer must be aware of these interactions and be careful about the sequence in which he makes design decisions. A particular method for treating the relationships between design variables is called "design strategy."

For many design problems, the truly creative part of the process seems to take place in the very early stages when the "design concept" is formed. This is the most abstract form of the solution, except, of course, for the more abstract functional descriptions.

Some design problems consist of complex sub-problems that require creative design, but for many, once the design concept is formed the solution is a relatively simple process of specifying details in such a way that the resulting solution meets the requirements of the problem. The fact that many computer instruction sets are so similar, suggests that the instruction set design problem is one for which most solutions can be generated by a single design concept.

This observation is the basis of an approach to writing a program that designs instruction sets. The trick lies in providing the program with an appropriate design concept that is general enough to include a broad range of instruction sets, but it must contain enough information to guarantee that the program can transform the concept into a solution in a reasonable amount of time.

## *GIS: A design concept for instruction sets*

Existing instructions for computers have many common features. A typical instruction occupies one word of the computer memory and consists of several fields of information, each encoded in a particular set of bits. Most computers have a field containing a code for an operation the computer is to perform when it executes the instruction. An instruction may be comprised of one or more fields containing addresses of locations in memory which embody information to be used during execution of the instruction. Some computers allow special methods, such as indexing and indirect addressing, for specifying data in the main memory of the computer. The purpose of GIS is to organize as many of these features as possible into a single, general model of a computer instruction set.

Since GIS is a model for a type of language, it can be described in the notation of Backus Normal Form. The complete description of GIS is rather detailed since it includes almost all of the features that have been used in instruction sets and a detailed description of the meaning of each syntactic feature of GIS.

For illustration, part of a GIS representation of an instruction is:

> <simple instruction> : : = <operation> <left operand part> <right operand part> <result part> <condition part> <if part> <else part>

The <operation> part of an instruction in GIS may be one of a list of 36 operations including:

> add, subtract, multiply, divide, compare, branch shift, move logical operations, and others.

The <left operand part>, <right operand part>, <result part>, <if part>, and <else part> are <addresses>.

An <address>, in turn, consists of many parts including displacement information, indexing, indirect addressing, bits to distinguish between references to various types of memory such as main memory or register memory, and other special techniques for specifying memory locations.

Each part of an instruction has an interpretation. The right and left operand parts specify operands which are to participate in the operation. The <result part> specifies an address where the result of an operation is to be stored. The <condition part> specifies some internal condition which may be set as the result of the operation. The <if part> specifies the address of the next instruction provided that the internal condition is satisfied and the <else part> specifies the address of the next instruction if the internal condition is not satisfied.

In most instruction sets, some of the GIS parts have implicit values. For example, in a single-address instruction format one of the operand addresses is always assumed to refer to the accumulator. The same is true of the result address. The if and else instruction addresses are assumed to refer to the next instruction in memory. To completely specify an instruction set by means of GIS, it is necessary to indicate whether each instruction part is implicit or explicit. The assumed value must also be specified for implicit parts while, for explicit parts, the parts of the instruction format used to encode the value of the part must be precisely specified.

GIS can be used to represent almost any instruction format in use in existing computers. From a syntactic point of view its primary limitation is its list of operations, which is necessarily restrictive since some operations in actual computers deal with special features and cannot be generalized. From a semantic point of view, GIS is not capable of all the subtle nuances assigned to certain instructions in some computers. For example, GIS makes no distinction between post-indexing and pre-indexing. In most cases, however, these subtleties have little effect on the design of the syntax of the instruction language which is of primary concern.

The most important attribute of GIS so far as the design program is concerned is that it is a design concept for instruction sets which it appears to represent at an appropriate level.

GIS meets the requirement of generality because it contains all the important addressing methods as special cases. It can be used to represent single address instructions, double- or triple-address instructions, memory-to-register instructions, and register-register operations, as well as others.

Another requirement is that a program using GIS as its model of an instruction set should be able, without a great deal of effort, to generate instruction sets that are plausible solutions to a design problem. GIS possesses this feature in the sense that any instance of the GIS model is indeed a valid instruction set.

## *ISDS: The design program*

The first step in the construction of ISDS was the selection of a method for storing GIS representations of instruction sets in the memory of a computer. The Backus Normal Form representation of GIS suggests a tree-like data structure. The structure actually used, called a "form-variable", is an IPL-V (Information Processing Language-V) list structure containing each instruction part identified by name and an attribute-value description list for each part to store important information about the part (whether it is implicit, whether the specification is a list of possible values or the number of bits needed to encode the time, and other descriptive information.)

All of the programs of ISDS are written in IPL-V, the primary reason being that IPL-V contains instructions for manipulating the tree-like data structure that is most appropriate for representing GIS instruction

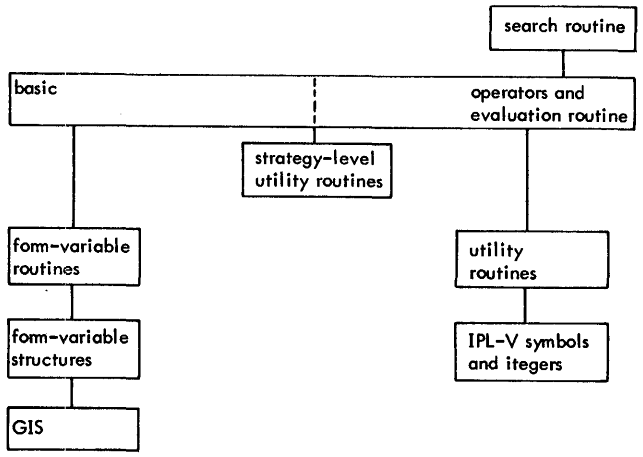

search routine
basic
operators and evaluation routine
strategy-level utility routines
form-variable routines
utility routines
form-variable structures
IPL-V symbols and itegers
GIS

Figure 1—Hierarchy of routines and data in ISDS

sets in the memory of a computer. However, the form-variable is a slightly more specialized data structure than the IPL-V list structure. Hence it was necessary to write a set of programs for manipulating form-variables.

These form-variable routines add items to form-variables, delete items, search for items, find attribute values on item description lists, and insert and delete attribute values on item description lists. The form-variable is a recursive data structure since an item may be a single value, a list of values or another form-variable.

The form variables of ISDS are at the lowest level of a hierarchy of routines (see Figure 1*) and are the building blocks of other routines in the sense that the higher-level routines make use of them to store new items in an instruction set, search for an item, and so on.

The form-variable routines are general in that they contain no information about instruction sets, GIS, or any aspects of the design process but are merely bookkeeping programs. ISDS contains another set of programs that are general in the sense that they perform the numerous computational tasks that must be undertaken during the design of an instruction set. These tasks include counting the number of items on a list and determining the number of bits required to encode a list of items.

At the level above the form-variable and computational routines, ISDS contains routines that add single parts to an instruction set. One such routine, for

* Figures 1 through 4 from thesis, "Using A Computer to Design Computer Instruction Sets", by Dr. Fred M. Haney. Carnegie-Mellon University.

example, adds a specified number of bits for designating index registers in a memory address. The number of bits is an input to this routine.

This routine performs no analysis, but merely the bookkeeping required to add a new part to an instruction set. The analysis required to determine the number of bits to be added for indexing is performed at the next level of the hierarchy. One routine, for example, adds indexing to the address references of an instruction set. For this routine the number of bits is not specified. The routine performs the analysis to determine the number of bits to be specified and then calls its counterpart which adds the specified number of bits. The routines which add specified parts to an instruction set are called "strategy-level utility routines". The routines which perform analysis and call for specific parts to be added are called "operators".

The routines in the higher level of ISDS are much more specialized than the low-level form-variable routines that can be used to represent many different kinds of objects. At the next higher level, the strategy-level utility routines are intended specifically for constructing instruction sets although they could be used in any design strategy since they have no decision power. Some decision power begins at the level of the operators which are based on a particular view of the relationships between the different parts of an instruction set. Each operator uses the values of certain parts

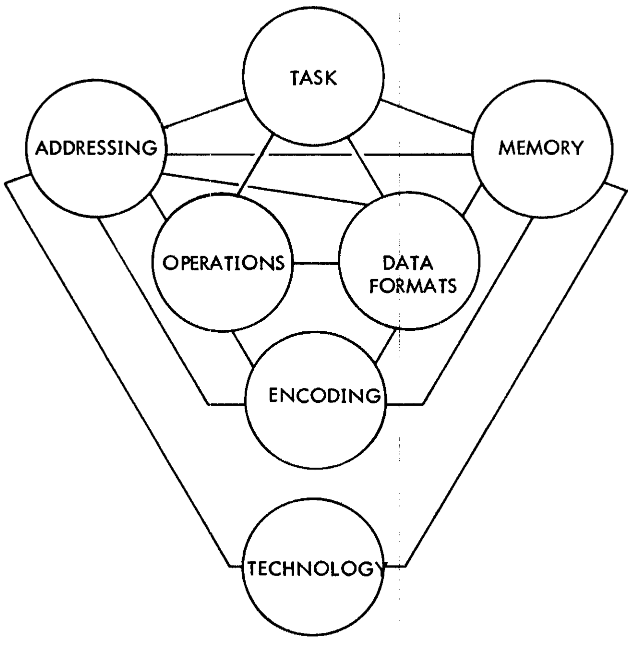


Figure 2—Relationships between the design variables

of the instruction set to determine the value of some new part. The types of possible relationships are illustrated in Figure 2.

In many cases, the relationship between parts of the instruction set are relatively obvious, but different results could be obtained with a different set of operators.

So far, nothing has been said about how the operators of ISDS are applied. One way is to write a program consisting of a sequence of calls on the operators. Operators that might be called, for example, are the address operator (which selects the number of addresses per instruction and the size of each address), the indexing operator, the indirect addressing operator, the arithmetic operator, and the logical instruction operator. (This program would be a specific design strategy for the instruction set design problem.) It must be recalled that a design strategy is a particular method for selecting the parts of a solution to a design problem. In particular, a design strategy is a specific choice of the independent variables that determine each part of the solution, together with a particular sequence in which the design decisions were made. As was pointed out, the operators represent a particular view of the independent variables and their influence on each part of the instruction set. The operators could have been used to write a set of different design strategies. Instead, however, a heuristic program that would determine its own strategy according to the demands of the design problem was written:

The statement of the design problem to this program consists of the following information:

1. An optional GIS representation of a particular instruction set containing features which must be included in the final product.
2. A cost-value matrix which assigns a relative cost and value to each instruction feature of GIS. The cost-value matrix also specifies a maximum cost for the instruction set.
3. Optional constraints on instruction features.
4. Memory size, word size, and byte size of the computer.

The heuristic design program consists of two routines; a basic strategy and a search routine. The basic strategy uses the memory size and word size to determine the number of addresses in each instruction and the general format of each address (whether it is a memory reference or an address augmented by a base register, page bit, etc.).

After this basic strategy has provided a starting point, the search routine adds one instruction part at

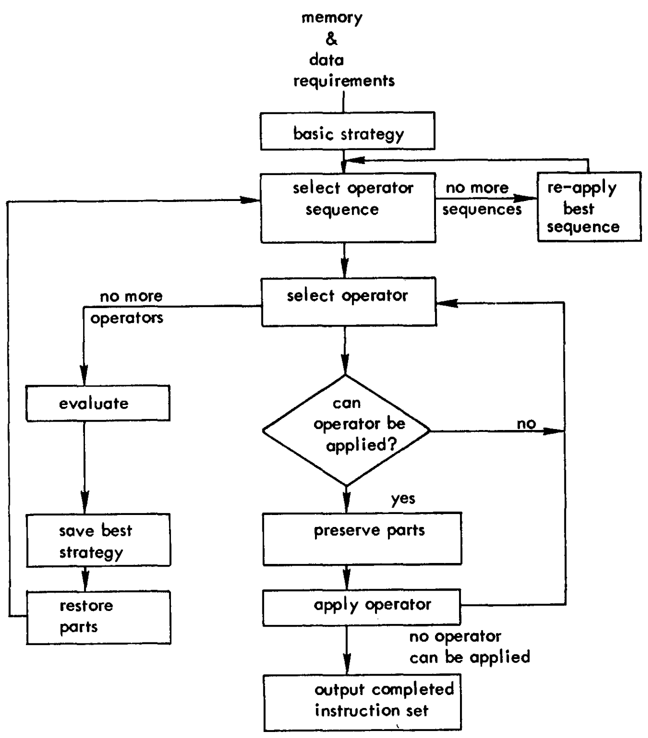


Figure 3—Optimization in ISDS

a time until there is no remaining space in the instruction format or the cost limit is reached. At each stage of the specification of the solution, the search routine tries every operator and evaluates the result with respect to the value coefficients provided in the statement of the problem (See Figure 3).

Corresponding to each operator there is a routine that restores the instruction set to its status before the operator was applied.

Hence, the sequence of events at each stage of specifications is "apply an operator", "evaluate", "restore", "apply the next operator", etc., until all operators have been applied, at which time the search routine reapplies the operator that resulted in the greatest improvement in the instruction set.

The search described above is a one-step search in the sense that the instruction set is evaluated after application of a single operator. Presumably much more interesting strategies could be obtained by evaluating after the application of sequences of operators, but the geometric increase in the computing time required made this approach impractical.

This example illustrates the operation of the heuristic program described above:

The following inputs were presented to the heuristic program:

1. A cost value matrix as follows:

| | *Cost* | *Value* |
|---|---|---|
| Indexing | 10 | 10 |
| Indirect Addressing | 0 | 20 |
| General Registers | 0 | 10 |
| Partial Word Address | 0 | 1 |
| Extra Operations | 0 | 1 |
| Permanent Adjustment To Index Registers | 0 | 10 |

2. A cost constraint of 10.
3. Required operations of add, subtract, multiply, divide, compare, and absolute value for fixed point and floating point arithmetic.
4. Required operations of "negate", "and", "or", and "no operation for logical data."
5. A move operation.
6. Memory size and word size of 65536 words and 36 bits respectively.

The basic strategy determines that 16 bits are required for each main memory address. Since five bits are needed to encode the required operations, there is only room in an instruction word for one address without some augmented addressing scheme. The basic strategy can specify augmented addressing, but for this case it specifies a single, main memory address specification of 16 bits. The search strategy specifies additional instruction features in the following sequence: general registers, indirect addressing, additional operations, additional operations, indexing, a permanent adjustment to an index register after indexing, operations, operations, partial word addressing. The resulting instruction set has the following format:

| 0 5 | 6 9 | 10 13 | 14 | 15 18 | 19 | 20 35 |
|---|---|---|---|---|---|---|
| Operation Code | Partial Word Address | General Register | Index Adjust | Index | Indirect Address | Memory Address |

This format is almost identical to the format of the Univac 1108 computer, however, the instruction set designed by ISDS is not. The primary difference is in the number of operations in the two instruction sets. The 1108 permits over 150 operations, whereas the ISDS instruction set contains only 52 operations.

The instruction sets also differ in their interpretation of some of the instruction features. However, this example shows that ISDS is capable of designing an instruction language that in its essential features resembles the instruction language of the Univac 1108.

It is interesting to note in the above example that if only 16 or fewer operations are required in the statement of the problem, then the basic strategy assigns four bits for the operation code and the remaining 32 bits permit two 16-bit memory references. In this case the search routine would not be able to apply any of the operators since every bit of the instruction word is used by the basic strategy. This illustrates a practical value of the present heuristic program; i.e., it permits a designer to learn by experimentation how the different design variables interact and how minor changes in one part affect the final product.

## SUMMARY

Working with ISDS indicates that for some design problems it is plausible to write programs that solve the design problem without human intervention. In general, the approach consists of the following steps:

1. Select a design concept—a model of solutions to the design problem.
2. Select a data structure for instances of the de-design concept.
3. Create operators that perform analysis and specify single parts of the model.
4. Create programs that use cost, value and constraint information from the statement of the problem to apply the operators in some sequence that results in a solution to the problem.

This process, as it is applied in ISDS, is illustrated in Figure 4.

To be of practical use, a design program based on the ISDS approach would require a more sophisticated search strategy than the one used in the present version of ISDS. In general, it is probably possible to find clever ways of selecting the operators to be applied without actually trying every one. Any such scheme would give the search much more direction and enable the program to evaluate strategies of depth greater than one.

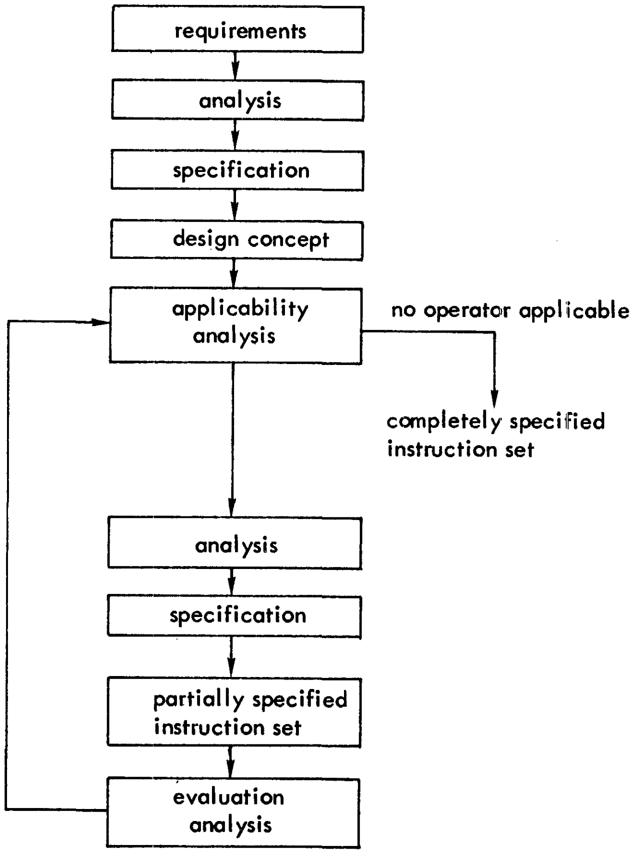


Figure 4—ISDS as a design model

The approach to automated design described is of limited use in many practical design problems. However, as designers experiment with interactive design systems they are likely to discover problems for which the so-called creative effort is relatively routine. For such problems, the approach of ISDS offers the prospect of more efficient automation than can be achieved in an interactive system.

## REFERENCES

1 M ASIMOW
*Introduction to design*
Prentice Hall, 1962 Englewood Cliffs N J
2 C ALEXANDER
*Notes on the synthesis of form*
Harvard University Press 1964 Cambridge Mass

# Directed library search to minimize cost

*by* DR. BRUCE A. CHUBB

*Lear Siegler, Incorporated*
Grand Rapids, Michigan

## *Statement of the problem*

The system engineer operating within the framework of a typical manufacturing organization operates from the following basic information and constraints:

a. A set of customer specifications to be met,
b. A basic system configuration to be used in realizing these specifications,
c. A set of standard components that fit into this configuration. The problem is to determine the collection of components that satisfies the given specification at *minimum total dollar cost.*

The above described situation exists in every area of system engineering where the configuration is "fixed" and a multitude of candidate components are available. The characteristics of these components can be stored in computer libraries by part numbers and an analysis program can be written to systematically analyze the system for any candidate set of components by merely inserting the appropriate part numbers. Such computer programs are structured so as to retrieve the data for each particular component, proceed with the various performance calculations and display the results to the designer for each set of part numbers manually selected.

This paper goes one step further and presents techniques and procedures for the effective use of computers in automating the solution to the above class of design problem.

## *Theoretical development*

### Development of the analysis program

The analysis section is the starting point of any computer-aided or automated design program. Optimization, in the design context, is derived from an efficient use of iterative analysis techniques. Devoid of a good analysis capability, the designer has nothing. Its presence provides a powerful tool in itself. In this case, however, it is simply a means to an end—*Automated Design.*

Although the internal details of the analysis program vary greatly for different applications, the input-output characteristics can be readily defined as shown in Figure 1. The first, and primary, requirement of the analysis program is that it must accurately represent

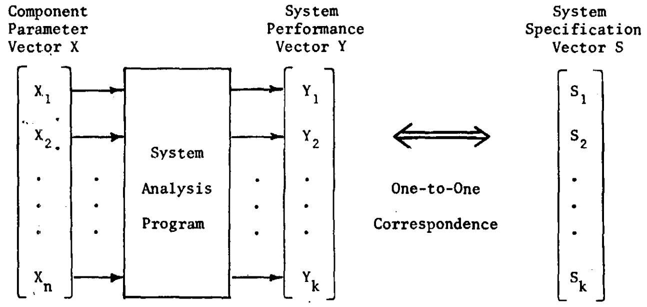

Component Parameter Vector X
System Performance Vector Y
System Specification Vector S
$X_1$
$X_2$
$X_n$
System Analysis Program
$Y_1$
$Y_2$
$Y_k$
One-to-One Correspondence
$S_1$
$S_2$
$S_k$

Figure 1—Input-output characteristics of system analysis program

the hardware. This requires a significantly detailed model, including often overlooked nonlinearities, and a realistic consideration of component tolerance effects. Second, the outputs of the analysis program must have a one-to-one correspondence with the list of system specifications. That is, if the customer specifies overshoot, response time, accuracy, etc., then the program must have the capability of calculating the system performance characteristics in this form. Third and last, since the analysis is to be repeated many times in an iterative fashion, the solution time should be a minimum.

The analysis problem is now defined mathematically by letting S, Y, and X be vectors, defined in general as:

System Specification Vector

$$S = [S_1, S_2, \cdots, S_k]$$

System Performance Vector

$$Y = [Y_1, Y_2, \cdots, Y_k]$$

Component Parameter Vector

$$X = [X_1, X_2, \cdots, X_n] \qquad (1)$$

where

k = number of performance specifications

n = number of component parameters

$S_i$ = numerical value for the ith specification $(1 \leq i \leq k)$

$Y_i$ = system performance function corresponding to ith specification $(1 \leq i \leq k)$

$X_j$ = numerical value for jth component parameter $(1 \leq j \leq n)$

Thus one can write in general that

$$\begin{bmatrix} Y_1 \\ Y_2 \\ \cdot \\ \cdot \\ \cdot \\ Y_k \end{bmatrix} = \begin{bmatrix} F_1(X_1, X_2, X_3, \cdots, X_n) \\ F_2(X_1, X_2, X_3, \cdots, X_n) \\ \cdot \\ \cdot \\ \cdot \\ F_k(X_1, X_2, X_3, \cdots, X_n) \end{bmatrix} \qquad (2)$$

where the F's represent the functions that need to be programmed to provide the system analysis. It is only necessary, at this time, that the X vector contain the elements as required to calculate the system performance function vector Y. However, it is convenient to include the component costs as part of the X vector [even though they will not appear explicitly in (2)] since they are required to calculate the optimization function that is introduced later.

Thus (2) can be used to calculate the system performance vector (Y) given any component vector (X). By programming this equation as presented, one obtains the desired analysis program except for one deficiency. That is, due to manufacturing tolerances, the X vector varies from unit to unit, and we are interested not in a particular value of Y but what spread or limits to expect. The tolerance effects can be included by using either the Monte Carlo or Moment Methods.[1,2] The latter technique is used in this paper since it also provides information that is extremely useful in minimizing the system cost.

The Moment technique makes use of an expansion of the function about the mean parameters using a Taylor series. The higher order terms of the series are neglected. This requires taking the partial derivative

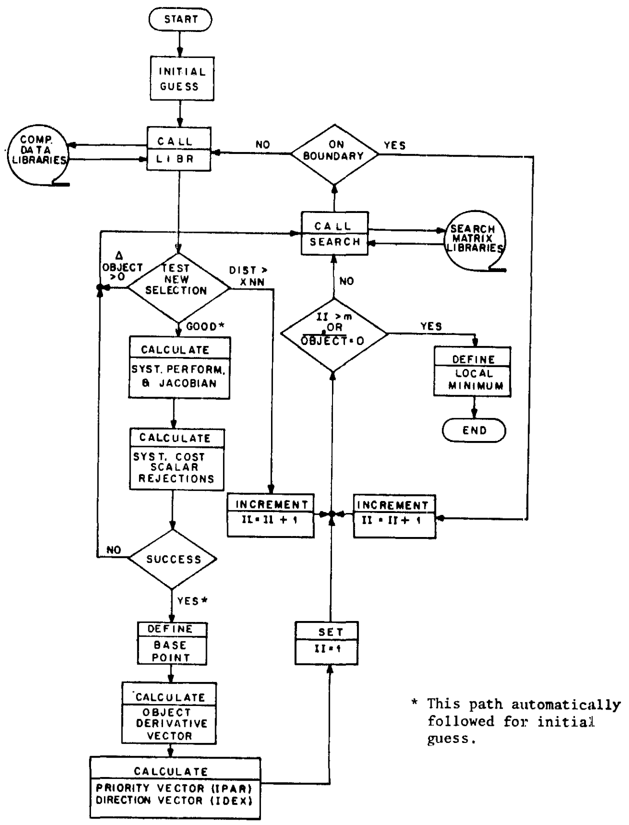


Figure 2—Computer aided design program flow chart

of each performance variable with respect to each component parameter. Assuming that the component performance parameters are independent and noting that the $\partial Y_i/\partial X_j = 0$ if $X_j$ is a component cost, the mean value of $Y_i$ is given by the equation

$$\mu_{Y_i} = F_i\,(\mu_{X_1}\,\mu_{W2}, \cdots, \mu_{X_n})$$

$$\sigma_{Y_i} = \sqrt{\left[(\sigma_{X_i})\frac{\partial Y_i}{\partial X_1}\right]^2 + \left[(\sigma_{X2})\frac{\partial Y}{\partial X_2}\right]^2 + \cdots + \left[(\sigma_{X_n})\frac{\partial Y_{iI}}{\partial X_n}\right]^2} \quad (4)$$

where $i = 1, 2, \cdots, k$ and the partial derivatives are evaluated while all other parameters are held at their mean value. As can be seen from (4), the use of the Moment method requires that we calculate the partial derivatives of each system performance function with respect to each component parameter. The matrix of these partials is the Jacobian.

$$J = \frac{\partial(Y_1, Y_2, \cdots, Y_k)}{\partial(X_1, X_2, \cdots, X_n)} = \begin{bmatrix} \frac{\partial Y_1}{\partial X_1} & \frac{\partial Y_1}{\partial X_2} & \cdots & \frac{\partial Y_1}{\partial X_n} \\ \frac{\partial Y_2}{\partial X_1} & \frac{\partial Y_2}{\partial X_2} & \cdots & \frac{\partial Y_2}{\partial X_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial Y_k}{\partial X_1} & \frac{\partial Y_k}{\partial X_2} & \cdots & \frac{\partial Y_k}{\partial X_n} \end{bmatrix} \quad (5)$$

The entries in the Jacobian are obtained numerically by programming (s) and using a subroutine to make the following steps:

1. Set all the $X_i$'s equal to their mean value $(\mu_{Y_i})$, and the calculated Y vector is taken to be the mean value $\mu_Y$.
2. $X_1$ is replaced by $(\mu_{X_I} + \Delta X_I)$ and the corresponding value of Y is calculated with all other X's at their mean value. From this, we obtain the first column of the Jacobian matrix using

$$\frac{\partial Y_i}{\partial X_j} \simeq \frac{Y_i - \mu_{Y_i}}{\Delta X_j} \text{ for } i = 1, 2, \cdots, k \text{ and } j = 1$$

3. Step 2 is repeated for each $X_j$ for $j = 1, 2, \cdots, n$ thereby obtaining the complete Jacobian matrix.

## Development of computer optimization design procedure

Use of the computer-aided design procedure described in the previous section, although many times more effective than any manual method, nevertheless represents only a passive use of the digital computer. That is, the engineer makes all the design decisions and the computer only serves as a fast calculator. The next logical step toward optimized design is to use the computer to determine how the components should be varied to converge on the desired minimum cost system.

Figure 2 illustrates in general how a computer could be used in a dynamic sense. The prerequisite to design is to input the data for all components. This is accomplished by loading in the component data cards prepunched in a prescribed format. This need be done only the first time and thereafter only if that data is to be changed; e.g., updated. These data are then stored by part number in an easily retrievable form on magnetic disk and are referred to as the "component libraries." In order to provide the mainline design program with a guide as to part number selection, some ordered array of these is desired. This is accomplished by using a "search matrix library," the precise working of which is explained later. Thus, immediately after generation of the component libraries, the computer calculates the component search matrices and stores these in a second block of data—the search matrix library. Now the program is ready to be used. The designer inputs the system specifications, fixed production labor costs, and any initial set of components of his choice. The latter item could be made a random selection if desired. In either event, the computer retrieves the component data from libraries and proceeds to calculate the system performance. The component parameters are then perturbated one at a time and the partials of each system performance function with respect to each component parameter are determined. Once this is completed the partials are stored in the form of a Jacobian matrix. The calculated performance limits are then compared to the specification limits. The fraction of the units produced that statistically fall outside of the specification limits is then calculated as the "rejection ratio." From this rejection ratio, the fixed labor cost, and the summation of the parts cost, the total cost is calculated. A printout is then made so that the user can follow the steps that a computer makes. Following this, some method must be employed to determine if cost is a minimum. If it is, then a final printout can be made. If it is not, then an option is shown as to how one wants to optimize. This can be accomplished by the user reading in another set of part numbers or the computer automatically can select a set in the manner described in a later section using the search matrix library. This procedure is repeated in an iterative manner until the optimum design is reached.

## Generation of object functions

The first question that must be answered in an optimization problem is, "What is to be optimized and what is optimum?" Often, this is not a trivial problem in itself since there are many separate and usually conflicting factors; i.e., minimum cost, maximum accuracy, small volume, best response, etc. These factors may be considered simultaneously be defining a scalar P of the form

$$P = \sum_{i=1}^{k} A_i(Y_i - D_i)^2 \tag{6}$$

where

- $P$ = object function to be minimized
- $k$ = number of desired properites
- $A_i$ = weight factor selected to give the $i^{th}$ property the desired priority
- $Y_i$ = current value of $i^{th}$ property
- $D_i$ = desired value for $i^{th}$ property

A serious difficulty inherent in this approach, however, consists in finding a set of weighting factors $A_1, A_2, \cdots, A_k$ such that scaling between the various terms is properly considered in order to maintain sensitivity and obtain good convergence. Considering properties such as accuracy, weight, cost and response, these weight selections often become subjective in nature.

It is proposed in this paper that an entirely different object function shall be used. It is founded on the competitive philosophy that the manufacturer wants a design that fulfills the customer requirements at minimum overall cost. With this result, he can either maximize his chances of competing or if his sale price is "fixed" he maximizes his profits. Using this minimum cost philsophy, an appropriate object function can be generated in the following manner.

The total cost to build a given number of systems is represented by the equation

$$\begin{matrix}\text{Total}\\\text{Cost}\end{matrix} = \begin{matrix}\text{Number}\\\text{Built}\end{matrix}\left[\begin{matrix}\text{Labor}\\\text{Cost}\end{matrix} + \sum \begin{matrix}\text{Component}\\\text{Costs}\end{matrix}\right]\left[1 + \begin{matrix}\text{Overhead}\\\text{Ratio}\end{matrix}\right] \tag{7}$$

However, the number that must be built for a given contract is given by

$$\begin{matrix}\text{Number}\\\text{Built}\end{matrix} = \frac{\text{Number Required}}{\left[1 - \begin{matrix}\text{Rejection}\\\text{Ratio}\end{matrix}\right]} \tag{8}$$

Thus, we have for the total cost

$$\begin{matrix}\text{Total}\\\text{Cost}\end{matrix} = \frac{\text{Number Required}}{\left[1 - \begin{matrix}\text{Rejection}\\\text{Ratio}\end{matrix}\right]}\left[\begin{matrix}\text{Labor}\\\text{Cost}\end{matrix} + \sum \begin{matrix}\text{Component}\\\text{Costs}\end{matrix}\right]\left[1 + \begin{matrix}\text{Overhead}\\\text{Ratio}\end{matrix}\right] \tag{9}$$

Since the number of required units and (1 + overhead ratio) are product terms which are not functions of the components, one obtains the same cost minimizing set of components using the function

$$\text{Cost} = \frac{\begin{matrix}\text{Labor}\\\text{Cost}\end{matrix} + \sum \begin{matrix}\text{Component}\\\text{Costs}\end{matrix}}{1 - \begin{matrix}\text{Rejection}\\\text{Ratio}\end{matrix}} \tag{10}$$

Equation (10) is the object function used for what is defined later as "the fine search mode." When it is at a minimum, the desired optimum set of components has been defined. However, one problem may exist in the early portion of the iteration cycle. That is, the design can be so far away from specification that, for all practical purposes, the rejection ratio is unity, the denominator of (10) goes to zero, resulting in infinite cost. As long as this occurs, (10) has no practical value. In fact, one loses all sensitivity in calculating partials, and there is no way of telling if one design is better than another. For this reason, a "course search mode" is defined. Its corresponding object function is:

$$Q = \sum_{i=1}^{k} A_i R_i (Y_i - S_i)^2 \tag{11}$$

where

$Q$ = object function to be minimized

$k$ = number of specifications to be met

$A_i$ = weight facgor for ith specification

$R_i$ = rejection ratio for ith specification

$Y_i$ = calculated system performance 3 sigma limit corresponding to ith specification

$S_i$ = ith specification limit

It should be further noted that

$Y_i = \mu_{Y_i} - 3\sigma_{Y_i}$ if $S_i$ is a lower limit, and

$Y_i = \mu_{Y_i} + 3\sigma_{Y_i}$ if $S_i$ is an upper limit.

Since Equation (11) is used only in the coarse search mode, selection of the weight factors is not too critical. For this study, $A_i$ was set at $1/S_i^2$ except for the case when $S_i$ equals zero and then $A_i$ was arbitrarily set equal to unity.

In the coarse search mode, cost is neglected in an attempt to determine the performance such that the rejection ratio becomes less than unity. The incorporation of the $R_i$ term in (11) greatly aids in the accomplishment of this condition. First it nulls each term in the summation which represents an overdesigned condition (i.e., $R_i = 0$) and secondly it applies a linearily increasing weight on the others according to their significance.

Once each of the $R_i$'s is driven less than unity, the cost becomes finite, and the optimization process is switched from the coarse to the fine search where (10) is used as the object function.

## Calculation of rejection ratio

The total rejection ratio R is the probability of a design falling outside of the specification, and assuming that the specification limits are constant, it is given by

$$R = 1 - \int_{L_{11}}^{L_{12}} \int_{L_{21}}^{L_{22}} \cdots \int_{L_{k1}}^{L_{k2}} f_{Y1,\,Y2,\,\cdots,\,Yk}(y_1, y_2, \cdots, y_k)\,dy_1\,dy_2 \cdots dy_k \tag{12}$$

where:

$$\left\{\begin{matrix} L_{i1} = -\infty \\ L_{i2} = S_i \end{matrix}\right\} \text{ for the i}^{\text{th}} \text{ specification an upper bound}$$

$$\left\{\begin{matrix} L_{i1} = S_i \\ L_{i2} = \infty \end{matrix}\right\} \text{ for the i}^{\text{th}} \text{ specification a lower bound}$$

The joint density of the Y's is given by:

$$f_{Y1,\,Y2,\,\cdots\,Yk}(y_1, y_2, \cdots y_k)$$

$$= \frac{e^{-[(Y - \bar{Y})M_Y^{-1}(Y - \bar{Y})^T]}}{(2\pi)^{k/2}\sqrt{|M_Y|}} \tag{13}$$

where:

$$(Y - Y) = [(y_2 - \mu_{Y1}),\ (y - \mu_{Y2}),\ \cdots,\ (y_k - \mu_{Yk})]$$

and the $(k \times k)$ covariance matrix $M_Y$ is

$$M_Y = JM_XJ^T \tag{14}$$

Since the component performance parameters are assumed independent and $\sigma_{Xi} = 0$ if $X_i$ is a component cost, one can write the component covariance matrix $M_X$ as

$$M_X = \begin{bmatrix} \sigma_{X_1}^2 & 0 & \cdots & 0 \\ 0 & \sigma_{X_2}^2 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \sigma_{X_n}^2 \end{bmatrix} \tag{15}$$

In order to evaluate R using (12), one must evaluate the multiple integral of dimension k. This can be accomplished using numerical techniques, however, the process is very time consuming. In the interest of minimizing computer time, one of the three alternate procedures listed in Table I are best implemented. Each of these approximations requires calculating only the individual specification rejection ratios ($R_i$ for i = 1, 2, ···, k) which are given by

$$R_i = 1 - \frac{1}{\sqrt{2\pi\sigma_{Y_i}^2}} \int_{L_{i1}}^{L_{i2}} e^{-\frac{1}{2}\left(\frac{y-\mu_{Y_i}}{\sigma_{Y_i}}\right)^2} dy \quad (16)$$

Equation (16) can be evaluated by using the standard error function

$$\mathrm{ERF}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-u^2} du \quad (17)$$

using the relationships summarized Table II.

Since the upper bound approximation is always on the safe side, it is the one used here. However, the independent approximation does lie between the two extremes and thus might be closer to the actual cases

## Object function derivatives

It is of necessity that the partial derivatives of the object function be calculated in .the steepest ascent method of optimization. If these derivatives were somehow known for the direct search technique, it would be of advantage since one could then conduct exploratory moves in descending order of importance. In our case, it would be a major task to perturbate each of the component parameters again and calculate the resulting change in the object function to obtain the partail derivatives. It is shown, however, that these can be obtained directly from the Jacobian matrix which is already available from the tolerance calculations; namely, Equation (5). This is accomplished in the following manner as derived first for the fine search and then for the coarse search.

The object function used in fine search, Equation (10), can be written as

$$C(X) = [K + f(X)]\,[1 - R(X)]^{-1} \quad (18)$$

where

| | | |
|---|---|---|
| X | = | component parameter vector $[X_1, X_2, \cdots, X_n]$ |
| C(X) | = | total system cost |
| K | = | labor cost |
| R(X) | = | rejection ratio |
| f(X) | = | $\sum$ component cost |

Taking the partial derivative of C with respect to $X_i$ and expanding to include all $X_i$

$$\left[\frac{\partial C}{\partial X_1}, \frac{\partial C}{\partial X_2}, \cdots, \frac{\partial C}{\partial X_n}\right] = \frac{1}{1 - R(X)}\left[\frac{\partial f}{\partial X_1}, \frac{\partial f}{\partial X_2}, \cdots, \frac{\partial f}{\partial X_n}\right] + \frac{K + f(X)}{(1 - R(X))^2}\left[\frac{\partial R}{\partial X_1}, \frac{\partial R}{\partial X_2}, \cdots, \frac{\partial R}{\partial X_n}\right] \quad (19)$$

Expanding the $\partial R/\partial X$ vector interms of the Jacobian defined by (3) one obtains the desired matrix equation for the fine search cost derivative vector as

$$\left[\frac{\partial C}{\partial X_1}, \frac{\partial C}{\partial X_2}, \cdots, \frac{\partial C}{\partial X_n}\right] = \frac{1}{- R(X)}\left[\frac{\partial f}{\partial X_1}, \frac{\partial f}{\partial X_2}, \cdots, \frac{\partial f}{\partial X_k}\right] + \frac{K + f(X)}{(1 - R(X))^2}\left[\frac{\partial R}{\partial Y_1}, \frac{\partial R}{\partial Y_2}, \cdots, \frac{\partial R}{\partial Y_k}\right] \begin{bmatrix} \frac{\partial Y_1}{\partial X_1} & \cdots & \frac{\partial Y_1}{\partial X_n} \\ \vdots & & \vdots \\ \frac{\partial Y_k}{\partial X_1} & \cdots & \frac{\partial Y_k}{\partial X_n} \end{bmatrix} \quad (20)$$

where:

$$\frac{\partial f}{\partial X_i} = \begin{cases} 1 \text{ if } X_i \text{ is a component cost} \\ 0 \text{ otherwise} \end{cases}$$

Table I—Estimates of total rejection ratio (R)

| Upper Bound | Lower Bound | Independent |
|---|---|---|
| $\sum_{i=1}^{k} R_i$ if $\sum_{i=1}^{k} R_i < 1$; 1 otherwise | $R_j$ where $R_j \leq R_i$ for all $1 \leq i \leq k$ | $1 - \prod_{i=1}^{k}(1-R_i)$ |

and the vector

$$\frac{\partial R}{\partial Y_1}, \frac{\partial R}{\partial Y_2}, \cdots, \frac{\partial R}{Y_k} \tag{21}$$

is referred to as the "rejection ratio derivative vector" and given the notation $\partial R/\partial Y$.

The calculation of the $\partial R/\partial Y$ vector, as required for the fine search mode, depends on the particular equation used in approximating the rejection ratio R [see Table I]. We consider here only the case where R is approximated by the upper bound [see Reference 3 for other cases]. Since in the fine search mode

$$\sum_{i=1}^{k} R_i < 1$$

one has

$$R(\text{upper bound}) = R_1 + R_2 + \cdots + R_k \tag{22}$$

and since $R_j$ is a function of $Y_i$ only for $i = j$

$$\frac{\partial R(\text{upper bound})}{\partial Y_i} = \frac{\partial R_i}{\partial Y_i} \text{ for } i = 1, 2, \cdots, k \tag{23}$$

and only the partials of the individual rejection ratios are required.

Considering the specification limit a constant, the magnitude of $\partial R_i/\partial Y_i$ is given by the $Y_i$ density function evaluated at the point $y_i = S_i$ and the sign of $\partial R_i/\partial Y_i$ depends on whether $S_i$ is an upper or a lower bound. That is

$$\frac{\partial R_i}{\partial Y_i} = \frac{\pm 1}{\sqrt{2\pi\sigma_Y{}^2}} e^{-\frac{1}{2}\left[\frac{S_i - \mu_{Yi}}{\sigma_{Yi}}\right]^2} \tag{24}$$

where

$S_i$ = ith specification limit

$\mu_{Yi}$ = mean value of $Y_i$ distribution

$\sigma_{Yi}$ = standard deviation of $Y_i$ distribution

and the + sign is taken if $S_i$ is an upper limit and the − sign is taken if $S_i$ is a lower limit.

The object function used for coarse search is of the form [see (11)]

$$F(X) = A_1R_1(X)[Y_1(X) - S_1]^2 + A_2R_2(X)[Y_2(X) - S_2]^2 + \cdots + A_kR_k(X)[Y_k(X) - S_k]^2 \tag{25}$$

Following the same type of procedure, as for the fine search, the coarse derivative vector is found to be

$$\left[\frac{\partial F}{\partial X_1}, \frac{\partial F}{\partial X_2}, \cdots, \frac{\partial F}{\partial X_n}\right] = 2\begin{bmatrix} A_1(Y_1 - S_1)R_1 + (Y_1 - S_1)^2 \frac{\partial R_1}{\partial Y_1} \\ A_2(Y_2 - S_2)R_2 + (Y_2 - S_2)^2 \frac{\partial R_2}{\partial Y_2} \\ \vdots \\ A_k(Y_k - S_k)R_k + (Y_k - S_k)^2 \frac{\partial R_k}{\partial Y_k} \end{bmatrix}^T \begin{bmatrix} \frac{\partial Y_1}{\partial X_1} & \frac{\partial Y_1}{\partial X_2} \cdots & \frac{\partial Y_1}{\partial X_n} \\ \frac{\partial Y_2}{\partial X_1} & \frac{\partial Y_2}{\partial X_2} \cdots & \frac{\partial Y_2}{\partial X_n} \\ \vdots & \vdots & \vdots \\ \frac{\partial Y_k}{\partial X_1} & \frac{\partial Y_k}{\partial X_2} \cdots & \frac{\partial Y_k}{\partial X_n} \end{bmatrix} \tag{26}$$

Equation (26) gives the desired partial derivatives of the coarse search object function with respect to each component parameter in the system. Again, like (20), it is in terms of the already available Jacobian matrix and no further parameter perturbations are required.

## Design program strategy

The design program developed as part of this study has two basic operating options—analysis and directed search. When operating with the analysis option, the component part numbers required for each analysis may be either read in from cards or selected at random by the program. In either case, as many consecutive runs are made as requested and a final printout is provided summarizing the best design obtained. Thus the engineer can make a rapid evaluation of a selected number of designs of his choosing, or, he can perform Monte Carlo runs by letting the computer select the part numbers at random.

With the directed search option, the computer program uses the object derivatives in connection with search matrices to direct the next component selection in an attempt to reduce the object function. This process is repeated in an iterative fashion until a local minimum is obtained. Since there is no guraantee that this condition is the absolute minimum, numerous starting points are employed and the one with the lowest cost in assumed to be the best design. The starting points for each search may be specified by the user or otherwise selected at random by the program.

The generation of the search matrices is a prerequisite to a directed search. A separate search matrix is used along with each component library and their generation automatically follows each library update. These matrices consist of an order array of the component part numbers defined by

$$S_i = \begin{bmatrix} s_{11} & s_{12} & \cdots & s_1 l \\ & & & \\ s_{21} & s_{22} & \cdots & s_2 l \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ s_{m1} & s_{m2} & \cdots & s_m l \end{bmatrix} \qquad (27)$$

where

$l$ = the number of parameters used to describe the i^th component

$m$ = the number of part numbers for i^th component stored in the library

$s_{nj}$ = a component part number for $1 \leq n \leq m$ and $1 \geq j \geq l$

Each column of $S_i$ corresponds to a particular parameter of the i^th component and the entries of the column consist of all the i^th component part numbers arranged in ascending order of the mean value of that parameter. That is, let the j^th column of $S_i$ correspond to the k^th component parameter of the X vector. Then $s_{1j}$, $s_{2j}$, $\cdots$, $s_{mj}$ are chosen such that

$$\overline{X_k(s_{1j})} \leq \overline{X_k(s_{2j})} \leq \overline{X_k(s_{3j})} \leq \cdots \leq \overline{X_k(s_{mj})} \qquad (28)$$

where

$\overline{X_k(s_{nj})}$ signifies the mean value of the component parameter $X_k$ for the part number stored in location $s_{nj}$

In order to explain the strategy used by the design program to conduct a search, the following definitions are established.

| | |
|---|---|
| *search* | = minimization process which begins with the initial set of part numbers and ends once a local minimum is found. |
| *base point* | = set of part numbers for which the object function is less than that calculated for any previous set of part numbers in a given search. |
| *sub-search* | = that part of a search which takee place between successive base points. |
| *exploratory move* | = a set of part numbers which are at least tentatively being considered for a system performance analysis. |
| *failure* | = an exploratory move which is analyzed and the object function obtained is greater than (o; equal to) that of the base point. |
| *success* | = an exploratory move which is less than that of the base point. |
| *local minimum* | = the object function corresponding to the base point which remains once all the exploratory moves analyzed in a given sub-search result in failure. |

Table II—Equations for calculating individual rejection ratios ($R_i$)

| | $S_i$ Upper Bound | $S_i$ Lower Bound |
|---|---|---|
| $S_i \geq \mu_{Y_i}$ | $0.5\left[1 - \mathrm{ERF}\left(\frac{S_i - \mu_{Y_i}}{\sqrt{2}\,\sigma_{Y_i}}\right)\right]$ | $0.5\left[1 + \mathrm{ERF}\left(\frac{S_i - \mu_{Y_i}}{\sqrt{2}\,\sigma_{Y_i}}\right)\right]$ |
| $S_i < \mu_{Y_i}$ | $0.5\left[1 + \mathrm{ERF}\left(\frac{\mu_{Y_i} - S_i}{\sqrt{2}\,\sigma_{Y_i}}\right)\right]$ | $0.5\left[1 - \mathrm{ERF}\left(\frac{\mu_{Y_i} - S_i}{\sqrt{2}\,\sigma_{Y_i}}\right)\right]$ |

Thus a search is made up of many sub-searches and each of the latter are in turn made up of numerous exploratory moves. Each exploratory move consists of changing one component part number while keeping the others fixed at the base point. Once an exploratory move results in "success," the move is defined as a new base point and new sub-search is started. This process is repeated until all the exploratory moves of a sub-search are exhausted and no success is found. The base point for this last sub-search defines the local minimum.

The following ten steps describe the general pattern of the program's search strategy:

1. The object function being minimized is Q [Equation (11)] while in the coarse search mode and COST [Equation (10)] while in the fine search mode. The program is in the coarse search mode as long as the total rejection ratio [Equation (12) or Table II] is equal to unity, once less than unity the program switches to the fine search mode.
2. Each time a lower object function is found, the corresponding part numbers are stored as a new base point.
3. At each new base point, calculations are made to establish the object function derivative vector using Equation (26) for the coarse search mode and (20) for the fine search mode.
4. Priority and direction vectors are established as the bases for making exploratory moves. The priority vector (IPAR) consists of a re-ordering of the component parameter numbers (i.e., subscripts of the X parameter vector) such that

$$\left|\frac{\partial\,\text{object}}{\partial X_{IPAR1}}\right| \geq \left|\frac{\partial\,\text{object}}{\partial X_{IPAR2}}\right| \geq \cdots \geq \left|\frac{\partial\,\text{object}}{\partial X_{IPARm}}\right| \quad (29)$$

where

m, the dimension of IPAR, equals the number of component parameters. The direction vector (IDEX) is defined by

$$\mathrm{IDEX}_{II} = \frac{\dfrac{\partial\,\text{object}}{\partial X_{II}}}{\left|\dfrac{\partial\,\text{object}}{\partial X_{II}}\right|} \quad \text{for } 1 \leq \mathrm{II} \leq \mathrm{m} \quad (30)$$

Thus

$\mathrm{IDEX}_{II}$ = +1 if the IIth parameter should be increased

= −1 if the IIth parameter should be decreased

in order to achieve a reduction in the object function.

5. A "sub-search progress number," denoted by the symbol II, is used by the program as the subscript for the IPAR and IDEX vectors. It is initialized equal to unity (i.e., II = 1) at the beginning of each sub-search and incremented under program control as the sub-search progresses. As II is increased from one to m, $\mathrm{IPAR}_{II}$ corresponds to the component parameter numbers having decreasing sensitivity values with respect to the object function. Likewise, $\mathrm{IDEX}_{IPARII}$ corresponds to the desired direction the $\mathrm{IPAR}_{II}$ parameter is to be changed.
6. Each exploratory move is initiated by calling a subroutine, named SEARCH, to select the new part number which is to be investigated. This is accomplished using the statement:

CALL SEARCH
[$\mathrm{IDEX}_{IPAR_{II}}$, $\mathrm{IPAR}_{II}$, $\mathrm{IPN}_{JJJ}$, IBOUND]

where

| | |
|---|---|
| $\mathrm{IDEX}_{IPARII}$ | = direction $\mathrm{IPAR}_{II}$ parameter is to be changed |
| $\mathrm{IPAR}_{II}$ | = parameter number for change being considered |
| $\mathrm{IPN}_{JJJ}$ | = present part number on entering the subroutine and on |

return it is the new part number to be used

| | | |
|---|---|---|
| IBOUND | = | 0 unless present part number is already at the boundary and cannot be changed further, then it is set to 1 by the subroutine |
| JJJ | = | component library number |

The SEARCH subroutine takes the $IPAR_{II}$ entry which corresponds to the subscript of the X vector and seeks the corresponding column of the appropriate search matrix. This column is then searched until the currently used part number is found ($IPN_{JJJ}$). Once this occurs the subroutine increments either down or up one location depending on whether IDEX is +1 or −1 and replaces the old part number with the new one found. If the old part number happens to be on a boundary such that a new part number cannot be obtained, the subroutine sets IBOUND to 1 and returns with the old part number. If this occurs, no further minimization can be obtained considering the $IPAR_{II}$ parameter, therefore one returns the part numbers to the base point and increments to the next most significant parameter by increasing the sub-search progress number (II) by 1 and step 6 is repeated.

7. For each new component selected by SEARCH a library subroutine, named LIBR, is called to retrieve the corresponding parameter data. This is accomplished by the statement

   CALL LIBR[$IPN_{JJJ}$, XMAX, XMIN]

   where

| | | |
|---|---|---|
| $IPN_{JJJ}$ | = | part number for which data is desired |
| XWMA | = | a vector containing the mean +3 sigma values for the total X parameter vector |
| XMIN | = | a vector containing the mean —3 sigma values for the total X parameter vector |

   The LIBR subroutine takes the part number ($IPN_{JJJ}$) and searches the appropriate component library stored off-line on magnetic disk, until the part number is located. Once located its associated parameter data is read back and inserted in the proper locations of the XMAX and XMIN vector. Thus by calling the LIBR subroutine with a part number, one is able to automatically update the three sigma limits for the X's corresponding to that part leaving the others unchanged.

8. After the new data is obtained for the exploratory move, the program checks for the existence of two conditions before the system performance is evaluated. The first is used to control the extent that the program explores changes based on a given parameter before it moves on to the next parameter. This is accomplished by calculating a normalized distance (DIST) according to

$$DIST = \frac{XMIN_i}{XMAXS_i} \quad \text{for } IDEX_i > 0$$

$$= \frac{XMINS_i}{XMAX_i} \quad \text{for } IDEX_i < 0 \qquad (31)$$

where $i = IPAR_{II}$

| | | |
|---|---|---|
| XMAXS | = | a vector containing the mean +3 sigma values for the total X parameter vector for the base point. |
| XMINS | = | a vector containing the mean —3 sigma values for the total X parameter vector for the base point. |

This normalized distance is then compared to a program input parameter XNN. For XNN > 1, one is assured that the $X_{IPARII}$ random variable has been varied so that its frequency distribution inside the 3 sigma limits lies outside the distribution for the corresponding base point parameter. Thus by selecting the value of XNN, the program user can control the extent to which exploratory moves are made. A value of XNN = 1.5 was found to give satisfactory results. By making XNN larger one explores more possibilities at the expense of increased computer time. Thus, for DIST < XNN the program returns the part numbers to the base point, increments to the next most significant parameter incrementing the sub-search progress number by one, and returns to step 6 above by calling SEARCH. If DIST ≤ XNN, the program continues to make the second check. This second check consists of calculating the estimated change in the object function based on its first derivative vector using the equation.

$$\Delta\text{object} = \sum_{i=1}^{m} \frac{\partial\,\text{object}}{\partial X_i}\left[\text{XNOM}_i - \text{XNOMS}_i\right] \quad (32)$$

where XNOM and XNOMS are the mean component parameter vectors corresponding respectively to the exploratory part number vector and the base part number vector. Since the i = $\text{IPAR}_{II}$ term in (32) is negative, one knows that if Δobject turns out to be positive, the summation of the changes caused by the parameters in $\text{IPN}_{JJJ}$ other than $\text{IPAR}_{II}$ have resulted in an estimated increase in the object function. Since an increase in Δobject is undesirable, one returns to step 6 above, when Δobject $>0$ and calls SEARCH keeping the same sub-search progress number (II). If Δobject $\leq 0$, a complete system performance analysis is made using the exploratory move part numbers.

9. If the exploratory move turns out to be "a success" (i.e., the object function is reduced) one returns to step 2 above and the process is repeated. If it is "a failure" (i.e., the object function isn ot reduced) one returns to step 6 and the next exploratory move is investigated.
10. The optimization procedure terminates once all the exploratory moves made from a given base point are completed "without success." This base point defines the local minimum.

Figure 3 summarizes the described design strategy in the form of a flow chart for the computer program. For simplicity sake, only the logic fundamental to the directed search option is included.

*Automated design example*

### Application problem

The example presented here is the automated design of an instrument servomechanism consisting of a follow-up device, electronic amplifier, drive motor with feedback generator, and geartrain. A pictorial diagram showing a fixed system configuration using these components is shown as Figure 4.

It is assumed that a design of this configuration must meet up to five preassigned specifications in the areas of damping, accuracy, and time response, Table III lists the specifications by name and vector notation, tells whether each specification is an upper or lower bound, and the units used.

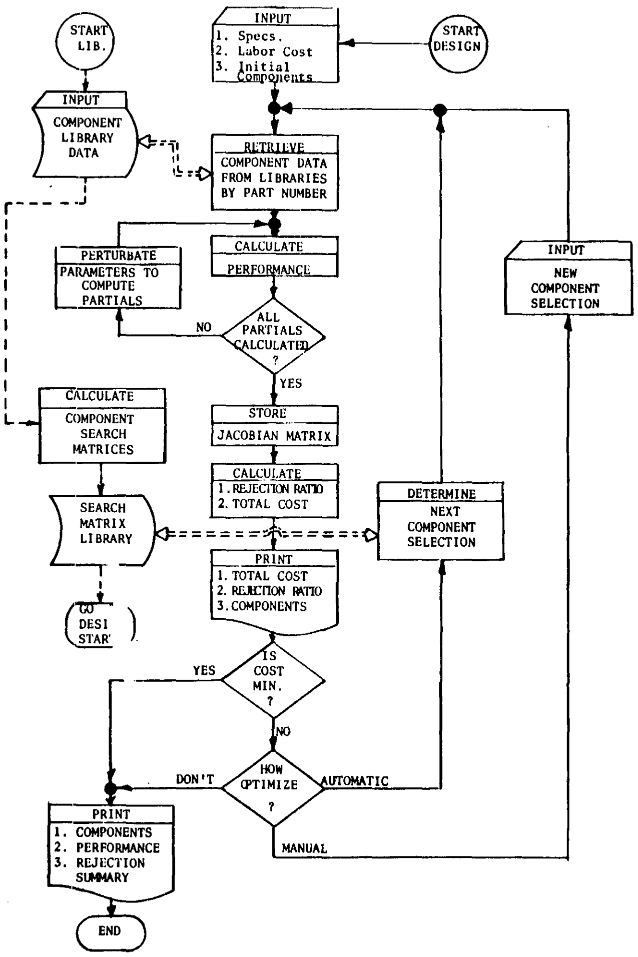

START LIB.
INPUT COMPONENT LIBRARY DATA
INPUT 1. Specs. 2. Labor Cost 3. Initial Components
START DESIGN
RETRIEVE COMPONENT DATA FROM LIBRARIES BY PART NUMBER
PERTURBATE PARAMETERS TO COMPUTE PARTIALS
CALCULATE PERFORMANCE
ALL PARTIALS CALCULATED ?
NO
YES
INPUT NEW COMPONENT SELECTION
CALCULATE COMPONENT SEARCH MATRICES
STORE JACOBIAN MATRIX
CALCULATE 1. REJECTION RATIO 2. TOTAL COST
DETERMINE NEXT COMPONENT SELECTION
SEARCH MATRIX LIBRARY
PRINT 1. TOTAL COST 2. REJECTION RATIO 3. COMPONENTS
TO DESI STAR
IS COST MIN. ?
YES
NO
HOW OPTIMIZE ?
DON'T
AUTOMATIC
MANUAL
PRINT 1. COMPONENTS 2. PERFORMANCE 3. REJECTION SUMMARY
END

Figure 3—Directed search basic program logic

Table III—System specifications

| Name | Symbol | Boundary | Units |
|---|---|---|---|
| Static accuracy | $S_1$ | upper | degrees |
| Resolution | $S_2$ | upper | degrees |
| Velocity lag | $S_3$ | upper | degrees |
| Follow-up rate | $S_4$ | lower | deg/sec |
| Damping ratio | $S_5$ | lower | - |

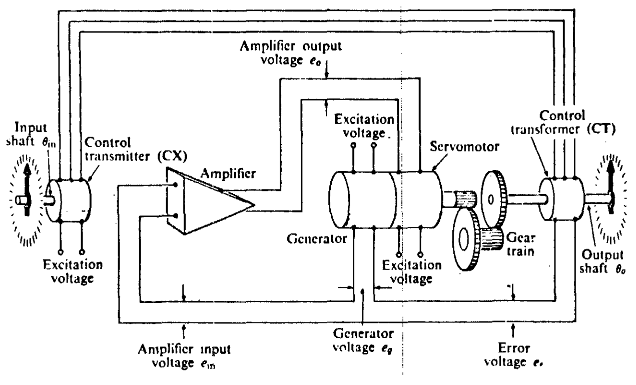


Figure 4—Schematic diagram of motor-generator instrument servomechanism

Table IV—System design equations

| Name | Symbol | Equation Used |
|---|---|---|
| Static Accuracy | $Y_1$ | $\theta_f + \frac{E_{an}}{K_f K_{af}} + \frac{K_{ag} E_{gn}}{K_f K_{af}} + \frac{E_s}{K_f K_{af}} + \frac{(T_g + T_\ell) E_c}{K_f K_{af} N T_s}$ |
| Resolution | $Y_2$ | $2 \left[ \frac{E_s}{K_f K_{af}} + \frac{(T_g + T_\ell) E_c}{K_f K_{af} N T_s} \right]$ |
| Velocity Lag | $Y_3$ | $\left[ \frac{N^2 (B_m + K_g K_{ag} T_s / E_c)}{K_f K_{af} N T_s / E_c} \right] \dot{\theta}_{in} + Y_1$ |
| Follow-up Rate | $Y_4$ | $\frac{\dot{\theta}_m}{N} \left[ 1 - \frac{T_g + T_\ell}{N T_s} \right] \frac{E_{sat}}{E_c}$ |
| Damping Ratio | $Y_5$ | $\frac{N^2 (B_m + K_g K_{ag} T_s / E_c)}{2 \sqrt{K_f K_{af} N T_s (N^2 J_m + J_g + J_\ell) / E_c}}$ |

Four component libraries are established to list the part characteristics as follows:

a. Follow-up—25 part numbers
b. Amplifier—50 part numbers
c. Motor-generator—25 part numbers
d. Geartrain—25 part numbers

Even though the size of each demonstration library was purposely kept small, the number of theoretical possible candidate systems is large; namely, 25 × 50 × 25 × 25 = 781,250.

The optimum collection of components is defined as "the one that satisfies the given specification in a manner resulting in minimum total cost."

## Component libraries and search matrices

The design equations corresponding to the five specifications are listed in Table IV [see Reference (4) for their derivation]. By grouping the parameters shown in Table IV according to component and adding the corresponding component cost, one obtains the X parameter vector as summarized in Table V.

In addition to specifying any desired combination of the above described five performance requirements, the user must a'so define the load that the servo is to drive. For the example program developed, the load is represented by an inertia ($J_\ell$) and a coulomb friction ($T_\ell$). These are shown as $X_{21}$ and $X_{22}$ of Table V.

The components selected to make up the libraries for this study, chosen so as to provide a broad base of design, are typical of those used throughout the servomechanism industry. An example of the parameter used is shown in Table VI which consist of the values follow-up component library.

Each column of the library data is 'abeled with the appropriate X-vector notation; i.e., $X_1$, $X_2$, $\cdots$, $X_{20}$, each of which is assumed to be a random variable with a normal distribution defined for each component by the mean ± 3 sigma limits given by the MAX and MIN values shown. The variables $X_i$ for i = 1, 4, 9, 16, 17, and 20, which are the individual component costs, motor rated voltage and the gear ratio and have no manufacturing tolerance, are still treated as random vairables" but having zero variance; $XMAX_i = XMIN_i$.

The search matrices are generated immediately after the library data is stored in the computer system. The search matrix for the follow-up is shown as Table VII and consists of the follow-up component part numbers arranged in an ordered array.

## Computer solution

In order to demonstrate the application of the program in its most comprehensive form, a customer requirement is assumed which makes use of all five specifications. The particular set is:

1. Static accuracy = 0.35 degrees
2. Resolution = 0.3 degrees
3. Velocity lag for 300 deg/sec input = 5 degrees
4. Follow = up late = 300 deg/sec
5. Damping ratio = 0.5

The assumed labor cost is $200.

The results obtained using the program in the direct search mode now are illustrated in detail for three searches. The first, shown in Table VIII, is a case where the initial guess fails completely to meet three out of

Table V—Component vector notation for library

| COMP | VAR | PARAMETER NAME | SYMBOL | UNITS |
|---|---|---|---|---|
| FOLLOWUP | $X_1$ | Cost | $C_f$ | dollars |
| | $X_2$ | Gain | $K_f$ | volts/rad |
| | $X_3$ | Accuracy | $\theta_f$ | minutes |
| AMPLIFIER | $X_4$ | Cost | $C_a$ | dollars |
| | $X_5$ | Gain to Followup | $K_{af}$ | volts/volt |
| | $X_6$ | Gain to Generator | $K_{ag}$ | volts/volt |
| | $X_7$ | Output Saturation Level | $E_{sat}$ | volts |
| | $X_8$ | Output Null Voltage | $E_{an}$ | volts |
| MOTOR GENERATOR | $X_9$ | Cost | $C_m$ | dollars |
| | $X_{10}$ | Stall Torque | $T_s$ | oz-in |
| | $X_{11}$ | No-Load Speed | $\dot{\theta}_m$ | rpm |
| | $X_{12}$ | Inertia | $J_m$ | gm-cm$^2$ |
| | $X_{13}$ | Starting Voltage | $E_s$ | volts |
| | $X_{14}$ | Generator Gain | $K_g$ | volts/1000 rpm |
| | $X_{15}$ | Generator Null | $E_{gn}$ | millivolts |
| | $X_{16}$ | Rated Control Voltage | $E_c$ | volts |
| GEAR TRAIN | $X_{17}$ | Cost | $C_g$ | dollars |
| | $X_{18}$ | Inertia | $J_g$ | gm-cm$^2$ |
| | $X_{19}$ | Friction | $T_g$ | oz-in |
| | $X_{20}$ | Gear Ratio | N | -- |
| LOAD | $X_{21}$ | Inertia | $J_\ell$ | gm-cm$^2$ |
| | $X_{22}$ | Friction | $T_\ell$ | oz-in |

Table VI—Followup library data

| | X VECTOR NOTATION | | | | |
|---|---|---|---|---|---|
| | $X_1$ | $X_2$ | | $X_3$ | |
| PART NO. | COST DOLLARS | FOLLOWUP GAIN (VOLTS/RAD) MAX. | MIN. | ACCURACY (MIN OF ARC) MAX. | MIN. |
| 1001 | 300.00 | 23.6000 | 21.4000 | 1.0 | 0.0 |
| 1002 | 24.00 | 12.7000 | 10.3000 | 10.0 | 0.0 |
| 1003 | 35.00 | 24.8000 | 20.2000 | 7.0 | 0.0 |
| 1004 | 200.00 | 0.5050 | 0.4950 | 30.0 | 0.0 |
| 1005 | 600.00 | 0.5025 | 0.4975 | 10.0 | 0.0 |
| 1006 | 28.00 | 24.8000 | 20.2000 | 15.0 | 0.0 |
| 1007 | 40.00 | 12.1000 | 10.9000 | 3.0 | 0.0 |
| 1008 | 36.00 | 12.1000 | 10.9000 | 7.0 | 0.0 |
| 1009 | 22.00 | 12.7000 | 10.3000 | 15.0 | 0.0 |
| 1010 | 30.00 | 0.5050 | 0.4950 | 120.0 | 0.0 |
| 1011 | 95.00 | 11.7000 | 11.3000 | 2.0 | 0.0 |
| 1012 | 90.00 | 0.5050 | 0.4950 | 60.0 | 0.0 |
| 1013 | 300.00 | 0.5050 | 0.4950 | 15.0 | 0.0 |
| 1014 | 60.00 | 24.8000 | 20.2000 | 3.0 | 0.0 |
| 1015 | 16.00 | 12.7000 | 10.3000 | 30.0 | 0.0 |
| 1016 | 30.00 | 25.9000 | 19.1000 | 10.0 | 0.0 |
| 1017 | 260.00 | 11.7000 | 11.3000 | 1.0 | 0.0 |
| 1018 | 150.00 | 23.6000 | 21.4000 | 2.0 | 0.0 |
| 1019 | 20.00 | 27.0000 | 18.0000 | 30.0 | 0.0 |
| 1020 | 28.00 | 0.5150 | 0.5050 | 180.0 | 0.0 |
| 1021 | 26.00 | 5.5000 | 4.5000 | 10.0 | 0.0 |
| 1022 | 30.00 | 5.2500 | 4.7500 | 5.0 | 0.0 |
| 1023 | 20.00 | 5.5000 | 4.5000 | 15.0 | 0.0 |
| 1024 | 28.00 | 5.2500 | 4.7500 | 7.0 | 0.0 |
| 1025 | 18.00 | 5.5000 | 4.5000 | 30.0 | 0.0 |

Table VII—Followup search matrix

| COST | K-F | THETA |
|---|---|---|
| 1015 | 1010 | 1017 |
| 1025 | 1005 | 1001 |
| 1023 | 1004 | 1018 |
| 1019 | 1012 | 1011 |
| 1009 | 1013 | 1014 |
| 1002 | 1020 | 1007 |
| 1021 | 1024 | 1022 |
| 1006 | 1021 | 1024 |
| 1024 | 1025 | 1003 |
| 1020 | 1022 | 1008 |
| 1022 | 1023 | 1021 |
| 1010 | 1015 | 1005 |
| 1016 | 1007 | 1002 |
| 1003 | 1011 | 1016 |
| 1008 | 1009 | 1013 |
| 1007 | 1017 | 1006 |
| 1014 | 1008 | 1023 |
| 1012 | 1002 | 1009 |
| 1011 | 1014 | 1015 |
| 1018 | 1006 | 1025 |
| 1004 | 1003 | 1019 |
| 1017 | 1018 | 1004 |
| 1013 | 1016 | 1012 |
| 1001 | 1001 | 1010 |
| 1005 | 1019 | 1020 |

the five required specifications, thus resulting in an infinite cost (shown as **** when cost $\geq$ 1. x $10^6$ dollars). Each line represents an analysis run and lists the cost (10), scalar (11), total reject [upper bound of (12)], the four component part numbers used, and the individual specification rejection percentages [$R_i$ using (16) for i = 1, $\cdots$, 5]. Fifty-five iterations are required by the program to minimize the scalar object function to the point where the cost becomes finite and the program switches from the coarse to the fine search mode. It should be noted that for this and subsequent computer runs, the intermediate printout is eliminated for all iterations where the scalar (cost when in fine search) is not reduced. These are considered "failure iterations" as is the case for numbers 2, 6, 7, etc., for the coarse search in Table VIII.

Once the program is in the fine search mode, the cost is minimized up to run number 202 where it is reduced from $38,261.30 to $374.27. As shown, an additional 23 iterations are required according to the termination procedure, as explained in an earlier section, in order to establish that part numbers 1009, 2003, 3002, and 4014 establish a local minimum.

Table IX illustrates the results obtained from the second search. This case represents the opposite condition where the initial guess at first hand looks like a "reasonable design"; i.e., the rejection is only 0.77 percent. However, after 74 iterations in the direct search mode, the cost has been reduced from the original design value of $555.30 to only $374.27—a savings of $181.03 per unit! The computer run time was less than one minute.

The third search is shown in Table X where this time the initial parts result in a design which fails completely to meet four out of the five specifications. After 55 iterations, the program has reduced the scalar from 59,610,000 to 3.396 and only one specification remains a complete failure; however, this point turns out to be a local minimum and no further reduction is obtained.

A total of 15 searches was made and the local minimums found and their frequencies are summarized in Table XI. Based on the results listed in Table XI, the system obtained using part numbers 1009, 2003, 3002 and 4014 is assumed to be the best design. The final computer printout sheet summarizing this combination is shown as Table XII.

Table VIII—Directed search with initial guess undersigned

| **********BEGIN SEARCH NUMBER 2 ********* | | | | | | | | | | | | | M |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RUN | | | PERCENT | COMPONENTS SELECTED | | | | *******INDIVIDUAL REJECTIONS***** | | | | | O D |
| NO. | COST | SCALAR | REJECT | FOUP | AMP | MOGEN | GRTR | STATIC | RES | LAG | FURATE | DAMP | E |
| 1 | ********* | 8.722E+02 | 100.00 | 1005 | 2046 | 3015 | 4022 | 100.00 | 100.00 | 100.00 | 0.00 | 0.03 | 1 |
| 3 | ********* | 7.147E+02 | 100.00 | 1005 | 2046 | 3012 | 4022 | 100.00 | 100.00 | 100.00 | 0.00 | 1.09 | 2 |
| 4 | ********* | 3.042E+02 | 100.00 | 1005 | 2046 | 3021 | 4022 | 100.00 | 100.00 | 100.00 | 0.0 | 100.00 | 2 |
| 5 | ********* | 8.662E+00 | 100.00 | 1021 | 2046 | 3021 | 4022 | 18.45 | 82.34 | 0.0 | 0.0 | 100.00 | 2 |
| 8 | ********* | 8.618E+00 | 100.00 | 1024 | 2046 | 3021 | 4022 | 4.18 | 82.88 | 0.0 | 0.0 | 100.00 | 2 |
| 12 | ********* | 8.611E+00 | 100.00 | 1022 | 2046 | 3021 | 4022 | 0.90 | 82.88 | 0.0 | 0.0 | 100.00 | 2 |
| 16 | ********* | 5.330E+00 | 100.00 | 1022 | 2046 | 3021 | 4024 | 0.01 | 21.95 | 0.0 | 0.00 | 100.00 | 2 |
| 18 | ********* | 4.497E+00 | 100.00 | 1022 | 2046 | 3012 | 4024 | 55.55 | 100.00 | 0.0 | 0.03 | 100.00 | 2 |
| 34 | ********* | 4.334E+00 | 100.00 | 1007 | 2046 | 3012 | 4024 | 0.0 | 0.00 | 0.0 | 0.03 | 100.00 | 2 |
| 35 | ********* | 3.359E+00 | 100.00 | 1007 | 2046 | 3023 | 4024 | 0.00 | 63.32 | 0.20 | 56.16 | 94.76 | 2 |
| 40 | ********* | 3.291E+00 | 100.00 | 1007 | 2046 | 3005 | 4024 | 0.00 | 74.21 | 27.87 | 88.46 | 55.68 | 2 |
| 41 | ********* | 3.207E+00 | 100.00 | 1007 | 2046 | 3011 | 4024 | 0.00 | 21.45 | 18.67 | 54.19 | 6.75 | 2 |
| 55 | 38261.30 | 3.222E+00 | 98.76 | 1011 | 2046 | 3011 | 4024 | 0.00 | 21.17 | 16.88 | 54.19 | 6.52 | 2 |
| 76 | 2989.50 | 3.213E+00 | 84.65 | 1011 | 2046 | 3011 | 4023 | 0.0 | 11.20 | 16.30 | 51.22 | 5.93 | 3 |
| 109 | 793.85 | 4.742E+00 | 28.32 | 1011 | 2025 | 3011 | 4023 | 0.0 | 0.0 | 0.00 | 28.32 | 0.0 | 3 |
| 119 | 565.04 | 4.254E+00 | 0.01 | 1011 | 2025 | 3011 | 4020 | 0.0 | 0.0 | 0.0 | 0.01 | 0.00 | 3 |
| 120 | 544.01 | 5.276E+00 | 0.00 | 1011 | 2025 | 3002 | 4020 | 0.0 | 0.0 | 0.00 | 0.00 | 0.00 | 3 |
| 122 | 537.01 | 5.349E+00 | 0.00 | 1011 | 2025 | 3002 | 4014 | 0.0 | 0.0 | 0.00 | 0.00 | 0.00 | 3 |
| 127 | 527.05 | 2.896E+01 | 0.01 | 1011 | 2041 | 3002 | 4014 | 0.0 | 0.0 | 0.01 | 0.0 | 0.0 | 3 |
| 133 | 527.00 | 1.636E+01 | 0.0 | 1011 | 2030 | 3002 | 4014 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 3 |
| 136 | 522.01 | 6.494E+00 | 0.00 | 1011 | 2033 | 3002 | 4014 | 0.0 | 0.0 | 0.0 | 0.00 | 0.00 | 3 |
| 149 | 487.00 | 5.766E+00 | 0.00 | 1011 | 2048 | 3002 | 4014 | 0.0 | 0.0 | 0.0 | 0.00 | 0.00 | 3 |
| 159 | 432.00 | 5.694E+00 | 0.00 | 1007 | 2048 | 3002 | 4014 | 0.0 | 0.0 | 0.0 | 0.00 | 0.00 | 3 |
| 162 | 428.00 | 5.450E+00 | 0.00 | 1008 | 2048 | 3002 | 4014 | 0.0 | 0.0 | 0.0 | 0.00 | 0.00 | 3 |
| 172 | 426.94 | 5.696E+00 | 1.16 | 1016 | 2048 | 3002 | 4014 | 0.0 | 0.0 | 0.0 | 0.00 | 1.16 | 3 |
| 178 | 424.28 | 5.522E+00 | 1.01 | 1006 | 2048 | 3002 | 4014 | 0.00 | 0.0 | 0.0 | 0.00 | 1.01 | 3 |
| 181 | 416.00 | 5.293E+00 | 0.00 | 1002 | 2048 | 3002 | 4014 | 0.00 | 0.0 | 0.0 | 0.00 | 0.00 | 3 |
| 184 | 414.00 | 5.144E+00 | 0.00 | 1009 | 2048 | 3002 | 4014 | 0.00 | 0.0 | 0.0 | 0.00 | 0.00 | 3 |
| 202 | 374.27 | 6.973E+00 | 1.41 | 1009 | 2003 | 3002 | 4014 | 0.00 | 0.0 | 1.40 | 0.00 | 0.00 | 3 |
| 225 | 374.27 | 6.973E+00 | 1.41 | 1009 | 2003 | 3002 | 4014 | 0.00 | 0.0 | 1.40 | 0.00 | 0.00 | 4 |

Table IX—Directed search with initial guess over*de*signed

| **********BEGIN SEARCH NUMBER 1 ********* | | | | | | | | | | | | | M |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RUN | | | PERCENT | COMPONENTS SELECTED | | | | *******INDIVIDUAL REJECTIONS***** | | | | | O D |
| NO. | COST | SCALAR | REJECT | FOUP | AMP | MOGEN | GRTR | STATIC | RES | LAG | FURATE | DAMP | E |
| 1 | 555.30 | 4.431E+00 | 0.77 | 1006 | 2050 | 3016 | 4013 | 0.04 | 0.0 | 0.73 | 0.00 | 0.00 | 1 |
| 2 | 550.65 | 4.030E+00 | 0.48 | 1006 | 2050 | 3002 | 4013 | 0.00 | 0.0 | 0.48 | 0.00 | 0.00 | 3 |
| 5 | 547.11 | 4.751E+00 | 0.02 | 1006 | 2050 | 3002 | 4020 | 0.00 | 0.0 | 0.0 | 0.0 | 0.02 | 3 |
| 7 | 544.01 | 3.936E+00 | 0.00 | 1006 | 2050 | 3002 | 4009 | 0.00 | 0.0 | 0.0 | 0.00 | 0.00 | 3 |
| 8 | 540.10 | 4.821E+00 | 0.02 | 1006 | 2050 | 3002 | 4014 | 0.00 | 0.0 | 0.0 | 0.0 | 0.02 | 3 |
| 12 | 535.89 | 4.537E+00 | 0.17 | 1002 | 2050 | 3002 | 4014 | 0.00 | 0.0 | 0.17 | 0.0 | 0.00 | 3 |
| 26 | 466.38 | 4.980E+00 | 0.08 | 1002 | 2025 | 3002 | 4014 | 0.00 | 0.0 | 0.08 | 0.00 | 0.00 | 3 |
| 27 | 464.79 | 4.890E+00 | 0.17 | 1009 | 2025 | 3002 | 4014 | 0.02 | 0.0 | 0.15 | 0.00 | 0.00 | 3 |
| 36 | 454.88 | 2.802E+01 | 0.19 | 1009 | 2041 | 3002 | 4014 | 0.00 | 0.0 | 0.19 | 0.0 | 0.0 | 3 |
| 42 | 454.00 | 1.550E+01 | 0.00 | 1009 | 2030 | 3002 | 4014 | 0.00 | 0.0 | 0.0 | 0.0 | 0.0 | 3 |
| 44 | 449.01 | 5.740E+00 | 0.00 | 1009 | 2033 | 3002 | 4014 | 0.00 | 0.0 | 0.0 | 0.00 | 0.00 | 3 |
| 56 | 414.00 | 5.144E+00 | 0.00 | 1009 | 2048 | 3002 | 4014 | 0.00 | 0.0 | 0.0 | 0.00 | 0.00 | 3 |
| 74 | 374.27 | 6.973E+00 | 1.41 | 1009 | 2003 | 3002 | 4014 | 0.00 | 0.0 | 1.40 | 0.00 | 0.00 | 3 |
| 97 | 374.27 | 6.973E+00 | 1.41 | 1009 | 2003 | 3002 | 4014 | 0.00 | 0.0 | 1.40 | 0.00 | 0.00 | 4 |

The validity that the above \$374.27 local minimum is also the absolute minimum can be checked, for this example, by using the procedure explained as follows: The lowest possible cost for a system made up of any collection of components is the summation of the individual component costs and the labor cost since if there are rejects, they only increase this cost. Therefore, to test if a local minimum is also the absolute minimum, one need analyze only the subset of the total combination for which

$$\text{labor cost} + \sum \text{component costs} < \text{local minumim} \quad (33)$$

If it turns out that analyzing each system in this subset

Table X—Directed search resulting in an unsatisfactory local minimum

| **********BEGIN SEARCH NUMBER 3 ********** | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RUN NO. | COST | SCALAR | PERCENT REJECT | COMPONENTS SELECTED FOUP | AMP | MOGEN | GRTR | *******INDIVIDUAL REJECTIONS***** STATIC | RES | LAG | FURATE | DAMP | MODE |
| 1 | ********** | 5.961E+07 | 100.00 | 1013 | 2014 | 3010 | 4017 | 99.91 | 100.00 | 100.00 | 100.00 | 0.0 | 1 |
| 2 | ********** | 4.616E+07 | 100.00 | 1013 | 2014 | 3014 | 4017 | 99.93 | 100.00 | 100.00 | 100.00 | 0.0 | 2 |
| 3 | ********** | 4.013E+07 | 100.00 | 1013 | 2014 | 3009 | 4017 | 99.94 | 100.00 | 100.00 | 100.00 | 0.0 | 2 |
| 5 | ********** | 2.078E+07 | 100.00 | 1013 | 2014 | 3004 | 4017 | 99.92 | 100.00 | 100.00 | 100.00 | 0.0 | 2 |
| 7 | ********** | 1.919E+07 | 100.00 | 1013 | 2014 | 3016 | 4017 | 99.89 | 100.00 | 100.00 | 78.15 | 0.0 | 2 |
| 8 | ********** | 1.812E+07 | 100.00 | 1013 | 2014 | 3002 | 4017 | 99.96 | 100.00 | 100.00 | 100.00 | 0.0 | 2 |
| 10 | ********** | 1.094E+07 | 100.00 | 1013 | 2014 | 3003 | 4017 | 99.97 | 100.00 | 100.00 | 100.00 | 0.0 | 2 |
| 11 | ********** | 6.726E+06 | 100.00 | 1013 | 2014 | 3011 | 4017 | 99.98 | 100.00 | 100.00 | 100.00 | 0.0 | 2 |
| 12 | ********** | 6.678E+06 | 100.00 | 1013 | 2014 | 3005 | 4017 | 99.98 | 100.00 | 100.00 | 100.00 | 0.0 | 2 |
| 13 | ********** | 5.485E+06 | 100.00 | 1013 | 2014 | 3023 | 4017 | 99.93 | 100.00 | 100.00 | 100.00 | 0.00 | 2 |
| 14 | ********** | 5.276E+06 | 100.00 | 1020 | 2014 | 3023 | 4017 | 99.95 | 100.00 | 100.00 | 100.00 | 0.00 | 2 |
| 15 | ********** | 5.687E+04 | 100.00 | 1024 | 2014 | 3023 | 4017 | 99.86 | 99.63 | 100.00 | 100.00 | 0.00 | 2 |
| 16 | ********** | 8.648E+01 | 100.00 | 1024 | 2014 | 3012 | 4017 | 99.82 | 56.60 | 100.00 | 100.00 | 0.0 | 2 |
| 17 | ********** | 3.746E+00 | 100.00 | 1024 | 2014 | 3017 | 4017 | 58.94 | 98.59 | 0.00 | 100.00 | 69.54 | 2 |
| 18 | ********** | 3.434E+00 | 100.00 | 1024 | 2014 | 3015 | 4017 | 26.65 | 47.15 | 0.08 | 100.00 | 0.00 | 2 |
| 19 | ********** | 3.399E+00 | 100.00 | 1022 | 2014 | 3015 | 4017 | 14.80 | 47.15 | 0.06 | 100.00 | 0.00 | 2 |
| 55 | ********** | 3.396E+00 | 100.00 | 1022 | 2006 | 3015 | 4017 | 14.80 | 47.15 | 0.06 | 100.00 | 0.00 | 2 |
| 156 | ********** | 3.396E+00 | 100.00 | 1022 | 2006 | 3015 | 4017 | 14.80 | 47.15 | 0.06 | 100.00 | 0.00 | 4 |

Table XII—Best design obtained using directed search

```
                        AUTOMATED DESIGN RESEARCH PROGRAM

                                JANUARY 15, 1969

                            ****DEFINITION OF LOAD****

                                       MAX              MIN
                  INERTIA (GM-CMSQR)   9.000E+02        7.000E+02
                  FRICTION (OZ-IN)     8.000E-01        4.000E-01

                     ****PART NUMBERS OF COMPONENTS SELECTED****

                     FOLLOWUP  AMPLIFIER  MOTOR-GEN  GEAR TRAIN
                       1009       2003       3002       4014

                                ****PERFORMANCE****

MAXIMUM     MINIMUM     SPEC LIMIT  PCT REJ
4.071E+03   2.928E+03                       TOTAL INERTIA  (GM-CMSQR)
4.578E+03   2.056E+03                       TORQUE CONSTANT (OZ-IN/RAD)
6.169E+01   2.717E+01                       DAMPING COEFFICIENT (OZ-IN-SEC)
4.895E+01   3.218E+01                       NATURAL FREQUENCY (HERTZ)
3.033E-01   4.712E-02      0.350     0.00   STATIC ACCURACY (DEG)
5.457E-02   2.666E-02      0.300     0.0    RESOLUTION (DEG)
5.295E+00   3.091E+00      5.000     1.40   LAG FOR 300. DEG/SEC RAMP (DEG)
9.702E+02   6.192E+02    300.000     0.00   FOLLOWUP RATE (DEG/SEC)
2.227E+00   1.233E+00      0.500     0.00   DAMPING RATIO



                               ****COST SUMMARY****

             1.41   PCT REJECTION (UPPER BOUND)
             1.41   PCT REJECTION (INDEPENDENT)
             1.40   PCT REJECTION (LOWER BOUND)
           200.00   LABOR COST
           169.00   PARTS COST
           374.27   TOTAL COST (USING R-UPPER BOUND)

            E SIGNIFIES CONVENTIONAL POWER-OF-TEN NOTATION
```

Table XI—Local minimums obtained for design example

| Number Times Occurred | System Cost | Rejection | Component Part Numbers | | | |
|---|---|---|---|---|---|---|
| | | | $P_1$ | $P_2$ | $P_3$ | $P_4$ |
| 1 | ∞ | 100% | 1016 | 2004 | 3024 | 4025 |
| 1 | ∞ | 100% | 1022 | 2006 | 3015 | 4017 |
| 1 | $410.01 | 0.25% | 1023 | 2008 | 3006 | 4014 |
| 1 | $394.63 | 1.43% | 1009 | 2003 | 3002 | 4002 |
| 1 | $394.29 | 0.07% | 1009 | 2012 | 3006 | 4014 |
| 9 | $374.27 | 1.41% | 1009 | 2003 | 3002 | 4014 |
| 1 | Search terminated as iterations exceeded maximum allowed of 300 | | | | | |

results in a total system cost higher than the local minimum being investigated, the latter is the absolute minimum.

For the above $374.27 local minimum there are 17,835 combinations which satisfy (33). This number although large is much less than the 781,250 total possible combinations and it becomes a practical value when one considers the solution time. The 17,835 combinations were, therefore, analyzed (at a cost of 1.7 hours of computer time compared to 74.4 hours for a complete exhaustive search) and each resulted in a total system cost > $374.27 thus proving the latter to be the absolute minimum.

## REFERENCES

1 D MARK
*Choosing the best method of variability analysis*
Electronic Design Nov 8 1963

2 D G MARK L H STEMBER JR
*Variability analysis*
Electro-Technology Vol 76 July 1965 35-48

3 B A CHUBB
*Computer aided optimization of nonlinear servomechanism employing a directed search of multiparameter component libraries and statistical tolerancing*
Michigan State Univ 1969 PhD Thesis

4 B A CHUBB
*Modern analytical design of instrument servomechanisms*
Addison-Wesley 1967

# Computer-aided design for custom integrated systems

*by* W. K. ORR

*The Singer Company-Friden Research Center*
Palo Alto, California

## INTRODUCTION

The computer-aided design (CAD) system described herein was developed to aid in the design of digital systems to be implemented by custom integrated circuits (CIC) and multi-chip hybrid custom integrated systems (CIS). The terms MSI/LSI are avoided here due to the general confusion which exists in the literature as to what constitutes an MSI/LSI circuit. The CAD system philosophy is that each CIC is implemented from a selected set of "library elements". This design approach results in some size inefficiencies, compared with manual designs, but provides many advantages, of which flexibility and a shortened design cycle are the most important. This CAD system captures fundamental design information in a machine-readable form early in the design process, thus maximizing potential computer assistance and minimizing costly and time-consuming errors. This paper contains an overview of the complete CAD system, highlighting its more distinctive features. The complete system has been operational on a 360/30 for several months, and specific experiences with it can therefore be discussed.

### *Overview*

Following are the major sections of the complete CAD system, and the distinctive features to be discussed more fully in later sections.

### Logic design

These programs convert a description of the logical function required of a CIS into the corresponding functional logic.

Distinctive features include:

1. Logicspec, a special register-transfer source language,
2. Compiled functional logic independent of hardware implementation,
3. Designer control of factoring and gathering.

### Logic simulation

These programs provide a complete simulated environment for the CIS, and a bit-simulation of response to input pattern sequences.

Distinctive features include:

1. Random access and cyclic memory,
2. Read-only-memory,
3. Time-dependent and conditional input signals,
4. Logic level statistics,
5. Selective output facilities.

### Logic conversion

These programs convert the functional logic to the logic family selected for hardware implementation, and create the design data-base.

Distinctive features include:

1. Efficient NAND/NOR logic generation,
2. Wired-OR

## Logic element design

These programs facilitate origination and revision of the library elements used in final system implementation:
Distinctive features include:

1. Graphospec, a special graphic source language,
2. Logic element library,
3. Artwork generation facility.

## Partitioning

These programs enable the designer to explore alternative partitionings, and post the final locations of all logical elements to the design data-base.
Distinctive features include:

1. Minimization of total pad-count for the CIS, and
2. Extensive designer/computer interaction.

## Element selection

These programs select the smallest eligible element meeting all the circuit requirements.
Distinctive features include:

1. Automatic insertion of gate expanders and intra-CIS pads, and
2. Capabilities for handling variable size elements.

## Element interconnection

This program establishes the X-Y interconnection routing.

Because of the nature of this paper, references are not cited in the text, instead an annotated bibiography is given at the end of the paper.

### *Logic design*

The initial input to the CAD system, as shown in Figure 1, consists of a set of Logicspec statements. Logicspec is a language which has been developed to simplify the task of describing a logic design in machine-readable form.

The Logicspec language permits the designer to avoid many of the burdensome details of logic design These details are filled in by the Logicspec Translator, which converts a Logicspec description into a complete set of design equations. These design equations are essentially Boolean equations, the operators being AND, OR and NOT. However, they are written in a modified form of polish notation. In this notation the equation

$$A = B \cdot C \cdot D + E \cdot F$$

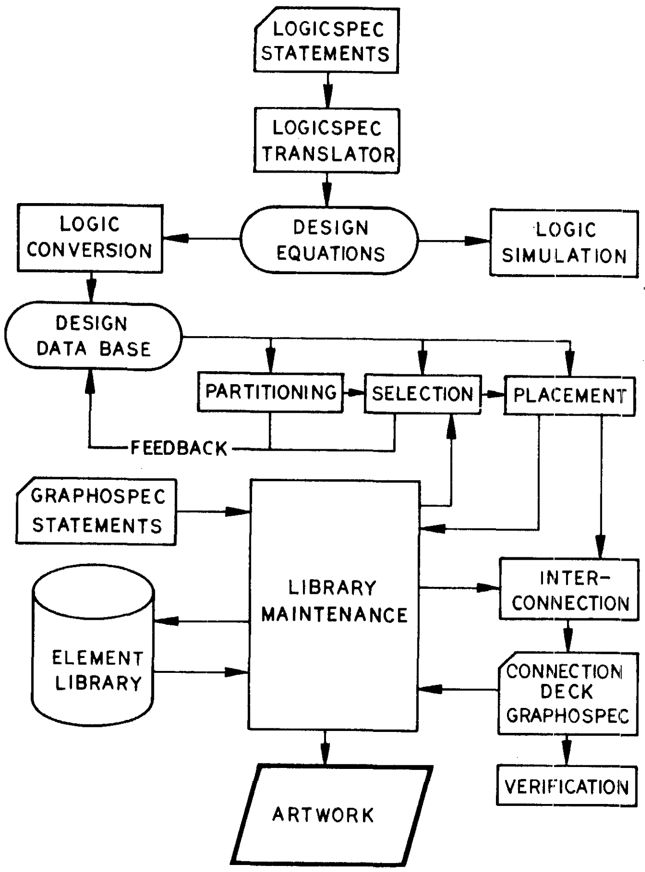


Figure 1—Computer-aided design system for CIS

appears as

$$A = ((B \quad C \quad D \cdot)(E \quad F \cdot) +)$$

An important characteristic of this notation is that each operator corresponds to a gate in an AND/OR implementation of the equation. This greatly simplifies those programs in the CAD system which must operate on these equations.

Since the Logicspec language is similar to other register transfer languages which have been proposed, only some of its more distinctive features will be discussed here; a full description will be published elsewhere.

Flip-flops are the only memory elements dealt with directly in a Logicspec description. Memory systems such as core and delay line memories are treated as systems interfaced to the logic design through signal lines. The description of these memories is deferred until simulation, where the simulator controller governs the manner in which the various memories interact via the signal lines, with the logic design.

Flip-flops are introduced in a description through the use of a Flip-Flop Collection declaration such as

FFC 12 A(1, 8*), B(1*, 8)

The foregoing indicates that the collections A and B both contain eight type-12 flip-flops. The "*" identifies the high order end of the collections for decoding references such as "A = 2". The type code ("12") is used by the simulation system to determine how the associated flip-flops are to be simulated and by other programs to determine how the flop-flips are to be implemented. This information is contained in an on-line disk library which can be expanded as required. Each flip-flop declared may have a maximum of five input and two output terminals:

A(1)/R, A(1)/S, A(1)/T, A(1)/P, A(1)/C, A(1) and A(1)′.

The functions of these terminals is determined by the information contained in the corresponding library entry.

The bulk of a Logicspec description consists of a set of statements which specify that if a certain *condition C* is true then an *action* S, or set of *actions* $S_1, S_2, \ldots, S_n$ occur.

The statement form actually used by the designer is the more concise conditional statement:

IF C THEN S

or when several actions are involved:

IF C THEN BEGIN $S_1$; $S_2$;...;$S_n$ END

The actions prescribed may include such operations as SET A, CLEAR B, C → D (transfer C to D), and INHIBIT TX. Conditional statements can be nested, i.e., $S_i$ could be another conditional statement. The condition C may be any Boolean expression formed using the operators + (OR), .(AND) and ′(NOT). It is permissible to describe an entire design using only Boolean equations; one need not use conditional statements if he so desires.

Most designers who have used the system feel that the conditional statement is rather cumbersome, and generally prefer to use an alternate form referred to as a qualification statement. This statement takes the form:

*C:

All subsequent statements are conditioned by C until another qualification statement occurs which overrides the condition C. To illustrate this consider the following:

```
FFC 12 A(1*, 3),  B(1*, 2), C(1*, 2);
      *(A = 3):     2 → B;
                    CLEAR C;
      *(A = 0):     B → C/S;
```

The modified Polish equations produced by the Logicspec translator for the above description are:

```
C(1)/C = *(A = 3)*
C(1)/S = (*(A = 0)* B(1)/1 · )
C(2)/C = *(A = 3)*
C(2)/S = ((*A = 0)* B(2)/1 · )
B(1)/R = *(A = 3)*
B(2)/S = *(A = 3)*
*(A = 3)* = (A(1)/0 A(2)/1 A(3)/1 · )
*(A = 0)* = (A(1)/0 A(2)/0 A(3)/0 · )
```

Qualification statements may be nested using a form of subscripting:

```
*C₁: S₁;
     S₂;
   *1 C₂:  S₃;
           S₄;
   *1 C₃:  S₅;
*C₄: S₆;
```

In the above, $S_1$ and $S_2$ are conditioned by $C_1$, $S_3$ and $S_4$ by $C_1 \cdot C_2$, $S_5$ by $C_1 \cdot C_3$ and $S_6$ by $C_4$ only. Logicspec is a free-form language, thus the identations above are for documentation only.

The structure of a Logicspec description contains important "clues" which are used by the translator to produce efficient logic. As an example, the majority of common control conditions are described using qualification statements. Referring to the above, $C_1$ is a common control condition in that it controls the actions $S_1$, and $S_2$ and in conjuntcion with $C_2$, $S_3$ and $S_4$. The Logicspec translator searches all qualification statements for such common conditions, and may either duplicate the gates involved every time the condition is used or generate a new signal which is used wherever the condition appears. This decision is under the control of the designer, who specifies the minimum number of times a condition must be used before a new signal is generated. The designer can also control the generation of new signals based on how the condition is used and the number of gates required to generate the condition.

The designer can use the flexibility described above to reduce the time required to simulate a design by instructing the translator to generate a new signal for every common condition. This generally reduces the number of gates in a design and thus the gate evaluation time during simulation.

The basic Logicspec language is very simple, but means are provided for extending the language through the use of subsystem definitions. A subsystem definition for the four bit ring-counter pictured in Figure 2 is given in Figure 3.

In Figure 3, line three is a signal collection declaration for the single rail bus OUT (double rail bus declarations begin with SIGC/2). Line four indicates that the words COUNT and SETO are to be added to the basic Logicspec vocabulary whenever a RINGC is used. Lines five and six simply describe fixed connections.

Once a subsystem has been defined and added to the subsystem library, the designer may use it in one of two ways—he may INCLUDE it or simply SIMULATE it as part of his design.

The INCLUDE option specifies that the actual text describing the subsystem is to be passed to the Logicspec translator and processed along with the text describing the rest of the design, in much the same way as a *macro* call functions in programming languages.

The SIMULATE option makes the logical description of the subsystem available for simulation purposes only—the rest of the logic design must interact with the subsystem through its input/output terminals. The subsystem logic does *not* become part of the system being designed: subsystem simulation information is passed directly to the simulation program, and is not processed by the Logicspec translator.

The same subsystem may be included and simulated in the same design. For example

```
INCLUDE RINGC A(AO), B(BO);
SIMULATE RINGC C(CO);
```

indicates that two ring-counters, A and B whose out put buses are AO and $BO_1$ respectively, are to be included in a design whereas C is only to be simulated.

The efficiency of the logic produced by the Logicspec translator has been evaluated, using designs for two systems which were in production before Logicspec was developed. These two systems were described in Logicspec, processed through the translator, and the resultant logic compared against that in the production systems. In both cases the logic produced by the translator contained five percent more gates than the production designs.

## *Logic simulation*

The electronics industry increasingly uses logic simulation to elimate logic design errors before com-

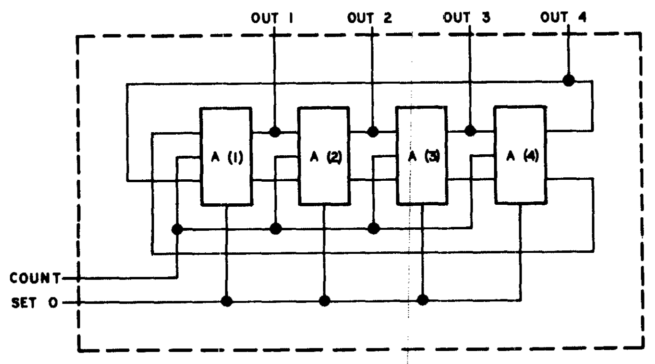


Figure 2—Four bit ring-counter

```
DEFINE RINGC (OUT);            (1)
FFC 12 A (1*, 4);              (2)
SIGC OUT (1*, 4);              (3)
OPERATION COUNT, SETO;         (4)
A -> OUT;                      (5)
A(4)' -> A(1);                 (6)
*SETO: CLEAR A;                (7)
*COUNT: SHR A;                 (8)
END;                           (9)
```

Figure 3—Subsystem definition for ring-counter

mitting a design to hardware. Many designers, however, insist on building breadboards to isolate lead-length and other circuit problems. In some cases, this is still a valid position. However, whenever the product will utlimately use CIC's a breadboard serves only to correct logic errors, simply because of the difference between the breadboard and final product technologies.

The creation of a logic simulation program begins with the simulator ordering program. This program orders the design equations, in preparation for the simulator compiler which produces the simulation code. The equation order, $E_1$, $E_2$,..., $E_n$, produced by the ordering program has the following property: the variable defined by equation, $E_i$, is a function of flip-flop outputs, system inputs (external inputs) or variables which have been defined in the preceding equations $E_1$,...,$E_{i-1}$. In addition a level list is produced which gives the number of gate delays in the definition of each signal. This list is used by the designer to isolate signal paths which contain excessive delays. These may be eliminated by changing the Logicspec description.

Whenever an equation occurs which defines a signal as a function of itself the program will fail to order it. At the completion of the ordering process a list of all unordered equations is produced. The designer must change his description such that every equation can be ordered before proceeding to the simulator compiler. From this the reader may wonder how flip-flops built from cross-coupled gates (latches) are processed. The answer is that the designer uses a flip-flop which has the characteristics of a latch, but he does not write the equations which describe the latch itself.

The simulator compiler generates code to evaluate each equation in the order specified by the ordering program. One pass through this code may represent one simulated clock time; the equivocation is clarified by the discussion of the simulator controller.

The simulator controller simulates all memory elements in a given CIS design, monitors various signals to find predesignated error conditions, and applies time-varying input signals so as to provide a realistic simulation of the environment in which the CIS must operate. A set of powerful commands has been developed to facilitate the designer's interaction with the simulator, and to maximize the information he receives about the simulation results. Concise statements are provided for describing wave forms which are to be applied to the machine's inputs (system inputs). Commands are provided to control the display of selected signals and flip-flops during simulation, as well as the status of any delay line or core memories involved in the design.

The flip-flop control procedure used by the simulator controller is outlined in Figure 4. A pass through the simulation code will define each signal and flip-flop input. If any asynchronous (non-clocked) flip-flop changes are required the controller makes these changes and another pass is made through the simulation code to propogate the effect of these changes. The controller counts the number of times recycling is required between clock times. If this count exceeds a limit specified by the designer, an error message is generated, thus permitting detection of any oscillating conditions which may be present in a given design. When there are no more asynchronous changes, a clock time is defined and all clocked flip-flop changes are made. This procedure for handling asynchronous flip-flop changes is also used to handle asynchronous changes in all other types of memories.

Simulation running time is clearly increased whenever asynchronous events occur. However, in the absence of asynchronous events there is virtually no run time overhead associated with the capability to handle such events. As regards running time, a logic system containing 100 flip-flops and 600 gates is simulated at a rate of 18 clock periods per second.

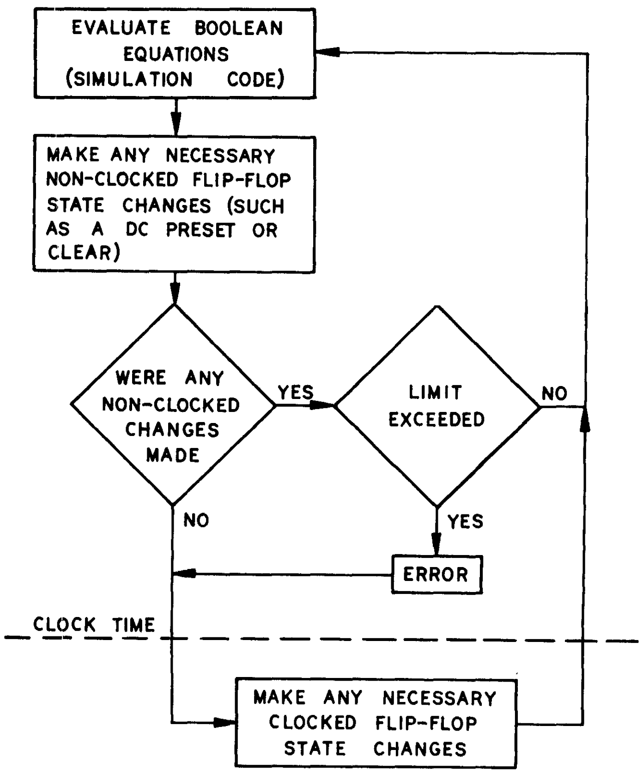

EVALUATE BOOLEAN EQUATIONS (SIMULATION CODE)
MAKE ANY NECESSARY NON-CLOCKED FLIP-FLOP STATE CHANGES (SUCH AS A DC PRESET OR CLEAR)
WERE ANY NON-CLOCKED CHANGES MADE
YES
LIMIT EXCEEDED
NO
NO
YES
ERROR
CLOCK TIME
MAKE ANY NECESSARY CLOCKED FLIP-FLOP STATE CHANGES

Figure 4—Simulator controller

*Logic conversion*

As discussed earlier, the logic produced by the Logicspec translator consists of a set of Boolean equations. Generally our logic is implemented in either NANDS or NORS, thus the design equations must be converted to one of these logic families.

The Logic Conversion Program is a one pass, table driven program capable of converting the design equations into either NANDS or NORS. When strapping (OR-tieing) is permitted, the program will use it when it yields a savings in gates and/or logic levels.

One of the unique features of this program is the order in which it converts the design equations. The conversion produced for the $i_{th}$ equation can be done efficiently (in terms of the number of gates required) only when it is known how the signal defined by this equation has been used—positively, negatively or both. In other words, to produce an efficient conversion for equation i one must first produce a conversion for each equation which uses the signal defined by equation i. On the surface this seems like a difficult problem, at least a time consuming task, however, as it turns out all of the necessary information is produced by the ordering program used in simulation.

Recall that the simulator ordering program produces the design equation ordering $E_1$, $E_2$,...,$E_n$, where every signal in equation $E_i$ has either been defined by a preceding equation or is a flip-flop output or system input. The conversion program converts the design equations in the order $E_n$, $E_{n-1}$,...,$E_1$. That is, the first equation converted is the one which appears at the end of the list produced by the simulator ordering program.

As the conversion is done, the program maintains a "usage list" which indicates how each signal has been used. As an example, if the equation $A = B + C$ is converted to NANDS the program records the fact that B and C have been used negatively, since the NAND conversion for this equation is $A = \overline{B} @ \overline{C}$, where @ represents a NAND gate. Thus, we see that when the program reaches equation $E_i$ the usuage list entry for the singal $V_i$ defined by $E_i$, contains all of the information as to how $V_i$ has been used. Returning to the previous example, if B was used only in the equation which defines A then the conversion program would produce an equation for $\overline{B}$ rather than B.

The table used by the conversion program to convert the design equations to NANDS, assuming a strapping capability, is shown below. This table is somewhat simpler than others which have appeared in the literature.

The entries in Table I give the NAND gate replacements for each Boolean operator as a function of the polarity that is required at a given level in the logic network. The "positive", "negative" entries which appear in the table are the polarities required on the inputs to the gate(s) which replace the Boolean operator. "Strap" implies that under the indicated conditions strapping may be used. Whenever the NOT operator occurs, it is simply removed with the indicated polarity reversal.

Figures 5a and 5b illustrate how the conversion table is used. In Figure 5a the implication is that a conversion is to be produced for H rather than $\overline{H}$; thus the first conversion table access is made with (Polarity, Boolean operator) = (POSITIVE, AND).

To insure that conversion is done correctly, the designer must supply a list of the system inputs and outputs with their required polarities. In addition, he must specify the polarity required at each flip-flop input.

Figure 7 shows a conversion produced for the design equations given in Figure 6. The symbol $ is used to indicate strapping. Each operator, @/$, is followed

TABLE I—NAND conversion table.

| | Boolean Operator | | |
|---|---|---|---|
| | AND | OR | NOT |
| POSITIVE | @@ (STRAP)<br>POSITIVE | @<br>NEGATIVE | ELIMINATE<br>NEGATIVE |
| NEGATIVE | @<br>POSITIVE | @@ (STRAP)<br>NEGATIVE | ELIMINATE<br>POSITIVE |

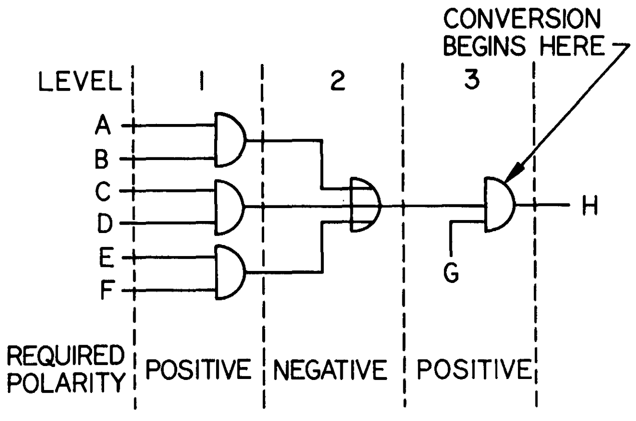


Figure 5a—H = (A · B + C · D + E · F) · G

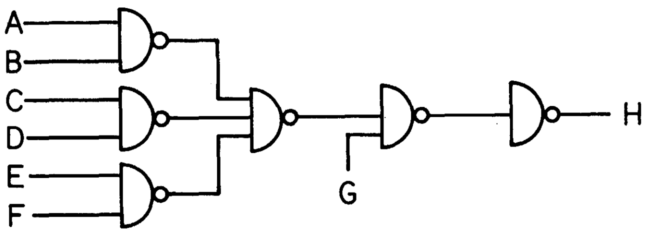


Figure 5b—NAND equivalent for H (without strapping)

by an operator number. The signal $\overline{Z}$, which appears as *Z′* in Figure 7, was produced rather than Z because this signal was described separately to the conversion program as a negative polarity system output.

Generally when AND-OR-NOT logic is converted to NAND or NOR logic, additional levels are introduced. The designer normally will pass the logic produced by the conversion program back through the ordering program to determine if excessive logic levels have been introduced. If there are excessive levels, the designer must eliminate them by changing the original Logicspec description.

To facilitate further processing, implementation equations such as those of Figure 7 are compacted into a file which resembles a wiring list. This file, referred to as the design Data Base, it used by all subsequent programs.

*Logic implementation*

Following logic conversion, artwork must be generated to produce the CIS which implements the logic contained in the design data base. In part this involves the selection of an IC equivalent for each gate and

```
1  Z =  ( ( (B A .) C(1)/1 +)' D(1)/1 .)

2  A =  ( (L K .)M +)

3  G = ( M N +)
```

Figure 6—Design equations

```
1        *M'* = ( M @10)

2        *N'* = ( N @11)

3 (1)    *Z'*=( ( ( B A @4) ( C(1)/1 @3) $2) D(1)/1 @1)

4 (2)    *A'* = ( ( L K @9) ( M @8)  $7)
5        A = ( *A'* @ 6)

6 (3)    G =  ( *M'* *N'* @5)
```

Figure 7—Implementation logic

flip-flop in the data base. The central information source used in establishing these equivalences is the Element Library. Since this library is used by all subsequent programs, it is appropriate to introduce it at this time.

The library elements important for the following discussion are gates (NAND/NOR), flip-flops, line drivers, and expanders. Although the Element Library contains a much broader range of digital elements, the CAD system is presently only capable of utilizing these simple logic elements to implement a CIS. The effectiveness of the CAD system will increase as the complexity and variety of library elements that can be used to implement a CIS is increased.

The library entry for each element contains all of the information required to produce the artwork for the several mask levels for the given element. This information is stored in a disc file in relocatable form so that an element may be positioned at any location on a chip in one of four possible rotations, and optionally as a mirror image. Dimensions, fan-in and fan-out capabilities and logic type are included in each element entry.

A complete set of programs accomplishes element library maintenance. Most important of these are the programs which the element designer uses in the creation and modification of library elements. Elements are generally built up in a bootstrap fashion. Resistors, diodes, and transistors are described to the system, in a special Graphospec language, as a collection of rectangles. In this language the description of a rectangle consists of the coordinates of one vertex, the length of the associated diagonal and a mask layer designation. More complex elements such as gates are described as collections of these elements. There is virtually no limit to the complexity of the elements that can be built up in this fashion.

The computing equipment currently available at the Research Center for use in CAD does not include a graphic display terminal. In anticipation that one will be available in the future the Graphospec language was designed for use on such a terminal.

*Partitioning*

The logic contained in the data base, representing that to be implemented by a CIS, may exceed the capacity of a single IC chip. It is then necessary to partition the logic into groups (partitions), each of which can be implemented by a single IC chip. It is important to note that partitioning is accomplished before IC equivalents have been selected for each gate and flip-flop in the data base. One reason for this is that there can be a significant size difference between gates and flip-flops whose inputs are generated and outputs used on the same chip and those whose inputs (outputs) originate (terminate) on a different chip. Thus, it is not possible to know exactly the area required for each element until partitioning has been done.

The approach taken toward partitioning was to develop a set of manipulation and reporting programs that the designer can put together to implement a wide variety of partitioning strategies. Some understanding of what is done by these programs can be gained from the following brief descriptions.

### The input program

Calculates approximate areas for each logic module. A given logic module consists of either a flip-flop, a collection of flip-flops, or the gates required to implement a design equation. As an illustration, the equation A = ( (B C @ 3) (D E @ 2) @ 1) is treated as a four input logic module which has one output cf. Figure 8. The area of this logic module is the sum of the approximate areas for the gates which make it up.

### The locate program

Places named logic modules on specified chips. Additionally, the designer can specify that a module is to be locked in place, that it cannot be moved from its designated location by subsequent programs. The name of a logic module is defined to be the name of the output signal (the name of the logic module in Figure 8 is A). Flip-flop outputs bear the name of the flip-flop.

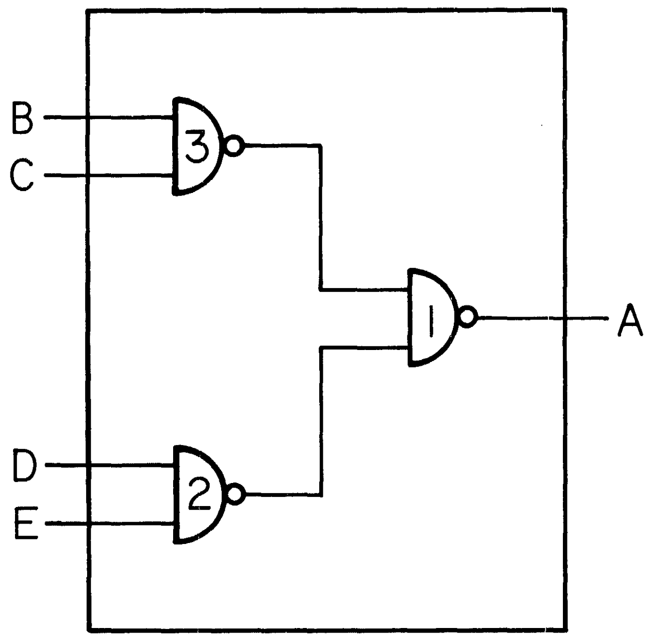


Figure 8—Logic module as defined for partitioning purposes

### The randomize program

Randomly distributes all logic modules which have not been placed over the chips which the designer designates as available. The designer can elect to begin partitioning with any number of chips.

### The weld program

Creates a new logic entity by associating any specified set of logic modules together. For example, one might weld the reset logic for a flip-flop collection to the flip-flop collection itself.

### The reduction program

Moves logic modules, or logic module sets, between chips whenever a move will result in a reduction in the total number of interconnection pads required within the CIS. Moves are made subject to the area and pad limitations the designer has given for each chip.

### The display program

Produces a chip interconnection table which gives the name of each "back-plane" signal, the number of the chip which generates the signal and the number(s) of the chip(s) the signal is connected to. This is only one of the several reports designed to aid the designer in executing his partitioning strategy.

Frequently an "optimum" partitioning job can be done only if the designer is willing to change his design. Gates can often be traded for pads, reducing system cost, also the duplication of registers, especially those that are extensively decoded, may reduce cost. The cost effectiveness of trade-offs such as these will of course change as packaging techniques improve, however, the situation will still arise when for want of a pad a chip must be added to a CIS.

To simplify the task of implementing design changes made to improve partitioning results, a facility is provided which allows the designer to obtain a "location deck" at any time during the partitioning process. Each card in this deck contains a logic module name and the number of the chip on which the module is located. The cards are punched in the format accepted by the locate program.

All design changes must be made to the associated Logicspec description—this fundamental design document is always kept up to date. Once a change has thus been made and a new data base created the location deck is processed to obtain the new partitioning results. If design changes eliminated certain logic modules the associated cards in the location deck are rejected. Further, if logic modules were added the designer is required to include cards for these in the location deck.

### *Element selection*

Chip locations for each logic module established by the partitioning process are posted to the design data base. The element selection program then selects the library elements that are to be used to implement the logic on each chip. Selection is controlled by a list, which the designer prepares, of eligible library elements. From this list, the program selects for each gate and flip-flop the element of smallest area which:

1. Provides the logic function required by the associated element in the data base,
2. Has the required fan-out capability, and
3. Has the required fan-in capability.

Whenever there is no eligible library element with adequate fan-in, gate input expanders are automatically added. Whenever the source and destination of a signal are on different chips, appropriate output and input pads are added automatically. To effect further area minimization, the selection program recognizes special combinations of logic elements and substitutes corresponding special library elements. At the moment the only special element substituted is a dual output NAND.

From this point on all processing is done on an individual chip basis.

### *Placement and interconnection*

Three layers of metal interconnections are generally required for the chips within a CIS. In such three layer systems the first metal layer is used solely for element *intra*connections, and the second and third layers are used for element *inter*connections. Thus, the CIS placement and interconnection task is equivalent to the two-sided PC card placement and interconnection task. The algorithms used are modifications of those which have proved effective tools for generating PC card artwork.

Element placement and interconnection are always done using the power bus, ground bus and pad layout prescribed by the designer. Several "standard" chip layouts are stored in the element library and the particular layout specified by the designer is referenced by the programs as required. A typical chip layout is shown Figure 9.

The CAD system can handle chips of various sizes, however there are certain aspects of chip layout which are standard from chip to chip:

1. Pads are located on the perimeter,
2. Power and ground busses are on separate metal layers—one under the other,

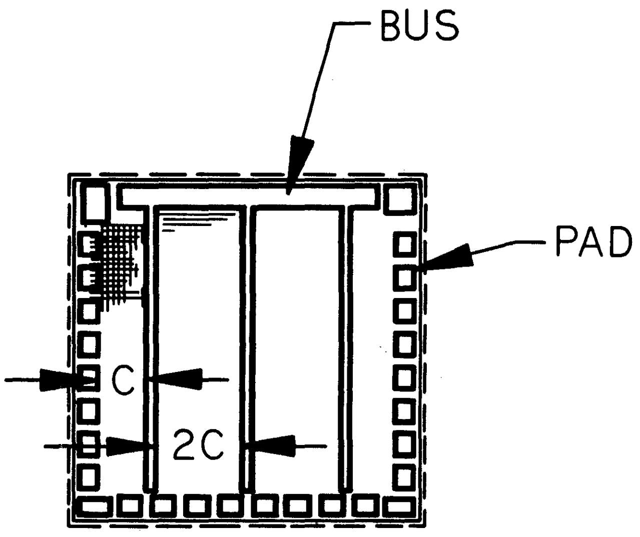


Figure 9—Typical chip layout

3. The minimum horizontal dimension of the region bounded by two segments of a bus or by a column of pads and a bus segment is C—the maximum must be 2C (cf figure 9).

The CIC placement problem is complicated by the fact that the library elements which must be placed on a chip are not all the same size. This is simplified somewhat by the restrictions imposed on chip layout and library element design. Reflecting the restrictions discussed regarding chip layout all library elements must be designed with one or the other of the aspect ratios pictured in Figure 10.

Placement is accomplished in three steps. First the elements and pads are placed on a regular grid, assuming that all the elements are the same size; the particular size chosen is that of the smallest element which must be placed on the chip.

Element pairs are then interchanged on this grid until a minimum approximate interconnection distance is found. Second, the elements are expanded to their full size into a new, initially empty, grid which actually represents the chip. Elements are processed one at a time starting at the center of the "small" grid and moving outward along a spiral path. For each element processed all possible positions on the new grid are evaluated with respect to three criteria: (1) the distance from the ideal position as defined by the small grid, (2) the degree of occupancy of this position by elements already processed, and (3) the angle of rotation between the lines defined by the grid center and ideal point and grid center and position being evaluated. The third criterion is designed to keep the expansion progressing outward from the center point. If the position picked as minimal with respect to the above criteria is partially or fully occupied, a search is entered to find other positions for the occupying elements. The third placement step is to again interchange pairs of elements so as to minimize interconnection distance, although this time only elements of the same size may be interchanged.

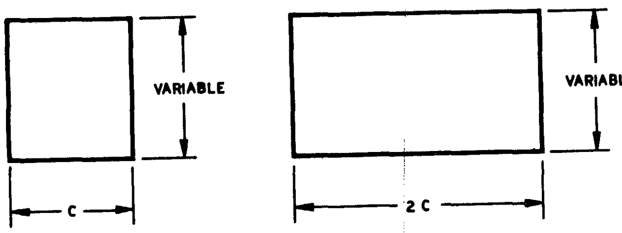


Figure 10—Permissible library element aspect ratios

For each chip processed, the placement program produces two outputs. The first includes a list of all of the library elements on each chip, with their absolute chip location given, this is entered in the element library. The second is a list of required interconnections; this is the input for the wiring program.

The wiring program makes all power and ground connections first, using a simple heuristic. Given that the point x, y is to be grounded or connected to power, a bi-directional search beginning at x, y is made in a direction perpendicular to the two closest segments of the appropriate bus. If an obstruction is encountered during the search a turn is made perpendicular to the preferred search direction. When one of the bus segments is encountered the required connection is made.

When all power and ground connections have been processed element interconnections are made using the Lee-algorithm. To speed up this process these connections are made in two steps. At first each pair of points to be connected is enclosed in a rectangle and the Lee-search is restricted to this enclosing rectangle. The particular rectangle chosen for a given pair of points is the one whose diagonal passes through the two points and is four units longer than the line joining the two points. If the program fails to make the connection within the enclosing rectangle the pair of points is added to a "failure list" and processing continues with the next pair. Once all point pairs have been processed pairs in the failure list are again processed; this time, however, the search area is not restricted.

Resticting the Lee-search as described above, in some cases improves running time as much as 28 percent.

The average density, in interconnections/square, of the chips processed to date has been 3.8, where a square is 10 wiring grid units on a side. At this density manual completion has been required for less than 1 percent of the interconnections processed.

To facilitate manual completion the output of the wiring program is a card deck referred to as a connection deck, which can be manually manipulated to make those connections which were not made automatically. These cards actually contain a description of the connections in the Graphospec language accepted by the element library maintenance programs. Thus, these programs can be used to plot metal masks, as a basis for deciding how to make the remaining connections.

This manual wiring completion procedure is a potential source of errors. A verification program is therefore provided to validate all manually introduced connections. Actually this program checks all connections in the connection deck against the design data base and produces an error list of all missing and erroneous connections. When a final connection deck is obtained it is entered in the element library. At this point the element library contains all of the information required to produce the artwork for a given chip.

## Non-recurring engineering costs

In June 1969 an experiment was performed to measure the non-recurring engineering costs of custom integrated circuit design.

For this experiment, a digital system whose logic design was already complete was chosen as a starting point. This system, as it existed in prototype form, consisted of 42 flip-flops, 215 NAND gates and 20 NOR gates implemented in 69 conventional dual inline packages.

The experiment began when the system design, in the form of four D-size logic diagrams, was received at the Research Center. Members of the CAD staff transformed the design into a machine-readable form using the Logicspec language. It should be noted that Logicspec was not being used as a design language, but merely as a means of conveying design information to the computer.

Following the transformation to Logicspec a complete logic simulation was performed to identify any errors introduced by the manual transformation. Several such errors were found. In addition, two errors were found in the logic diagrams.

At the beginning of the experiment it was decided that the CIS would be implemented using chips measuring 140 mils on a side with a maximum of 39 signal pads/chip. On these chips power and ground buses and pads occupied approximately 7,100 mils$^2$ of the available area, leaving 12,500 mils$^2$ for the placement of library elements.

Using a parts list submitted with the logic diagrams it was estimated that with the selected chip size the system could be implemented using six chips. The six chip partition obtained is characterized below. The area utilization figures given below were obtained after element selection had been performed.

The area utilization figures given in Table II clearly indicate that the six chip partition makes somewhat inefficient use of the available area. At the time it was not obvious that fewer chips could be used due to pad limitations. For this reason the experiment was completed using the six chip partition. Subsequently a five chip partition was obtained this is characterized below.

The five chip partition required more gates than the six chip partition because it was necessary to trade gates for pads in order to stay within the prescribed pad limits.

Following element selection each of the six chips was processed through the placement and wiring programs. Of the six chips processed only one required manual completion:one connection was made manually.

The end product of the experiment was a complete set of rubylith mask masters (11 mask layers) for one chip (chip five). In determining costs it was assumed that the mask masters for each of the remaining chips would cost approximately the same.

The professional manpower and computer costs required to perform the experiment are summarized below. At the Research Center all plotting is done in a multiprogramming environment (i.e., it is overlapped); for this reason the summary is broken into two parts. The entire experiment was completed in an elapsed time of three weeks.

The non-overlapped time shown above was the time

TABLE II—Six chip partition.

| Chip | Gates | Flip-Flops | Pads Used | Area In Mils$^2$ | % of Available Area |
|---|---|---|---|---|---|
| 1 | 26 | 9 | 31 | 10,762 | 86% |
| 2 | 40 | 7 | 38 | 11,343 | 90% |
| 3 | 38 | 9 | 38 | 11,438 | 91% |
| 4 | 32 | 4 | 37 | 7,446 | 59% |
| 5 | 43 | 6 | 37 | 11,052 | 88% |
| 6 | 28 | 7 | 36 | 8,696 | 69% |
| Total | 207 | 42 | 217 | 60,757 | |
| Average | 34 | 7 | 36 | 10,126 | |

TABLE III—Five chip partition.

| Chip | Gates | Flip-Flops | Pads Used | Area In Mils$^2$ | % of Available Area |
|---|---|---|---|---|---|
| 1 | 33 | 10 | 37 | 12,321 | 98% |
| 2 | 44 | 7 | 36 | 12,117 | 97% |
| 3 | 37 | 10 | 39 | 12,136 | 97% |
| 4 | 51 | 8 | 38 | 12,117 | 97% |
| 5 | 45 | 7 | 39 | 12,162 | 97% |
| Total | 210 | 42 | 189 | 60,853 | |
| Average | 42 | 8.4 | 37.8 | 12,170 | |

TABLE IV—Manpower and computer costs for experiment (exclusive of plotting).

| | Professional Man Hours | % of Total | Computer Hours (IBM 360/30) | % of Total |
|---|---|---|---|---|
| Transfer Design to Logicspec | 40.5 | 39% | 1.8 | 7% |
| Logic Simulation | 26.0 | 25% | 1.6 | 7% |
| Convert to Nand Logic | 2.0 | 2% | 0.4 | 2% |
| Partition System to 6 chips | 31.0 | 30% | 11.0 | 45% |
| Library Element Selection | 0.5 | 0.5% | 0.6 | 3% |
| Placement of Elements | 3.0 | 3% | 4.1 | 17% |
| Interconnection of Elements | 1.0 | 1% | 4.5 | 19% |
| Totals | 104.0 | | 24.2 | |

TABLE V—Computer costs for plotting portion of experiment.

| | Computer Time (IBM 360/30) Non-Overlapped | Overlapped (Plotting) |
|---|---|---|
| Prepare Composite for Manual Interconnection Completion | .3 | .8 |
| Prepare Mask Masters Chip 5 | 3.3 | 11.8 |
| Prepare Mask Masters for Other Chips (Extrapolation) | 16.5 | 59.0 |
| Totals | 20.1 | 71.6 |

required to load a disk file with the information which was to be plotted.

Experience at the Research Center indicates that approximately 2.5 nonprofessional man hours are required to strip and check a rubylith mask master. Including this the total cost for a final set of mask masters for the six chips is as summarized in Table VI.

TABLE VI—Non-recurring engineering costs of CIS design.

| | |
|---|---|
| Professional Man Hours | 104.0 |
| Non-Profesional Man Hours | 165.0 |
| Non-Overlapped IBM 360/30 Hours | 44.3 |
| Overlapped IBM 360/30 Hours | 71.6 |

## CONCLUSION

Development of the CAD system described herein required approximately twelve man years of effort. The system is now providing the tools which make the task of developing a CIS as simple as, and as regards non-recurring costs, no more expensive than the task of developing the same system using discrete IC packages and printed circuit boards.

## ACKNOWLEDGMENTS

The author is indebted to Messers. L. P. Robinson and G. Hare for their continued support and encouragement. The development and implementation of the system was carried out by the author, J. Landau, L. P. Robinson, R. Quick, A. Watson, and V. Wilson. We are all greatful to those designers who struggled with it during its infancy.

## REFERENCES

An excellent source paper on computer-aided design is:
1 M A BREUER
*General survey of design automation*
Proc IEEE Vol 54 1966 1708-1721

The basic CAD System philosophy is similar to Motorola's Polycell approach
2 M S CALLAHAN
*Moving into MOS production*
Electronic News Vol 4 1968

A classical paper on register transfer languages is:
3 D F GORMAN J P ANDERSON
*A logic design translator*
Proc FJCC 1962 86-96

The modified polish notation is discussed in:
4 W K ORR J M SPITZE
*Design automation utilizing a modified polish notation*
Proc FJCC 1964 643-650

A good description of equation ordering techniques appears in:
5 I H YETTER
*High-speed fault simulation for Univac 1107 computer system*
Proc ACM Nat Conf 1968 265-277

The simulator compiler was patterned after:
6 R A RUTMAN
*LOGIK a syntax-directed compiler for computer bit-time simulation*
Masters Thesis Univ of Caif at Los Angeles 1964

A logic conversion technique similar to the one described appears in:
7 M KLERER G KORN
*Digital computer user's handbook*
McGraw-Hill Book Co Inc N Y 1967 4-185-4-192

The exchange algorithm used in placement is similar to:
8 J POMENTULE
*An algorithm for minimizing backboard wiring functions*
CACM Vol 8 1965 699-703

The wiring algorithm used appears in:
9 C Y LEE
*An algorithm for path connections and its application*
IRE Trans on Electronic Computers Vol 10 1961 346-365

# An overview of the computer output microfilm field

*by* DON M. AVEDON*

*Scan Graphics Corporation*
Stamford, Connecticut

## INTRODUCTION

From the earliest times, man has made his mark. At first his marks were made with his own fingers on walls of caves. He used a chisel or brush to create pictures of animals. He developed symbols, alphabet and languages. Man used marks to pass information from person to person and from generation to generation. Through the ages, man recorded information to be used again and again. He recorded history, mathematics and law. These things brought order to his life. The history of civilization is the history of man's ability to communicate, record and make marks.

In making marks, there is most always a moving object. Man used his own fingers. Today most marks are made by a type slug, a print hammer, a moving drum, or some mechanical device. And now man has electronic digital computers. These machines manipulate and generate information at unprecedented speed. Man's need to make marks has multiplied many times in the past few decades. Much of the drudgery of handling information has been relegated to the computer. The speed of computers is so great that mechanical mark-making devices can no longer keep pace. Devices using a stylus or print hammer will not move fast enough and require too much maintenance.

This is the beginning of our story—a new method for making marks—COM.

What does COM mean?

1. Computer Output Microfilm: microfilm containing data, produced by a recorder from computer generated electrical signals.
2. Computer Output Microfilmer: a recorder which converts data from a computer into human readable language and records it on microfilm.
3. Computer Output Microfilming: a method of converting data from a computer into human readable language onto microfilm.

This paper will describe COM technology and the various types of COM recorders. Some of the uses and applications will be explored. A description of the various recorders and a comparison of the units will be made. Microfilm origination, dissemination and retrieval systems will be reviewed. Some COM market forecasts will be looked at and a survey of the field by the National Microfilm Association will be presented.

### *General*

Over the past several years, American industry as well as the scientific community have turned increasingly to the use of computers *and* microfilm as a means of controlling what is referred to as the "paperwork explosion." Computers and microfilm have been generally used independently to cope with the same problem. Both have been successful, but neither alone has completely solved the problem. The effect of combining microfilm and the computer in a system for information handling may turn out to be more dramatic than the effect of either alone.

Computer systems of all generations, first, second and

* Also Director, National Microfilm Association, Annapolis, Maryland.

third, have been plagued by an imbalance of speeds. The functions of computer systems namely, input processing, and output—though intertwined as functions, have been sadly imbalanced in their speed relationships one to the other. The computer itself, or the main frame, has seen an ascension of speed and power of phenomenal proportions from the mid-1950's to the present. The older vacuum tube equipment could process at thousandths of seconds or milliseconds. The transistor and solid state technology brought forth microseconds or a millionth of a second speeds. Finally, the third generation in this evolution, the micrologic of integrated circuits, has caused nanosecond speeds, a billionth of a second, to be realized. However, the input/output twins have seen no similar evolution. On the input side the basic medium of data input is still the EAM card which is over 30 years old. On the output side mechanical printing and its hardcopy paper medium has been the major avenue of getting the information to the user.

Although there have been several major efforts to improve the input/output situation, and especially to eliminate the output bottleneck, none has succeeded until now. The Computer Output Microfilmer, or COM recorder provides the solution to the computer output problem. A COM recorder has the output equivalent of as many as 30 impact printers operating simultaneously. Some COM units have a transfer rate as high as 100,000 characters per second (transfer rate: the speed at which information can be transferred from magnetic tape to microfilm).

The COM is a device which records computer data on microfilm in human readable form. It is a recorder which may be connected directly to the computer for "on line" operation or to a magnetic tape unit for "off-line" operation. The magnetic tape unit "reads" information into the COM from a magnetic tape which previously has been recorded directly from the computer.

There are three types of COM devices:

Business—alphanumeric printer

Scientific—alphanumeric printer and plotter

Graphic Arts—special quality alphanumeric printer and plotter

Recording the output of a digital computer directly on microfilm is not new. As early as 1955 at least one COM recorder was in use for this purpose. The early units as well as some of the new units were designed for scientific work. These recorders are printer-plotters; that is, they are capable of reducing the digital output of computers to convenient, usable plots and curves that are annotated with alphanumeric information. Figures 1 and 2 are typical scientific plots. This was the role of the COM until recent years when some of the scientific users began using the printing capability for non-scientific alphanumeric listings.

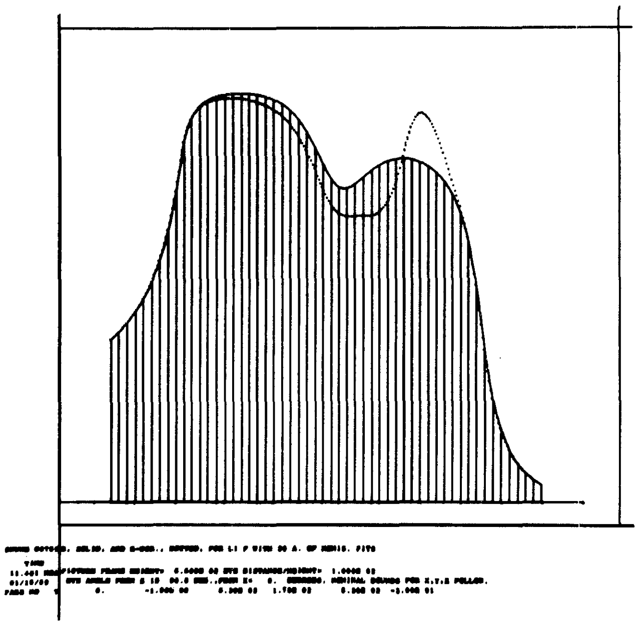


Figure 1—Typical scientific plot

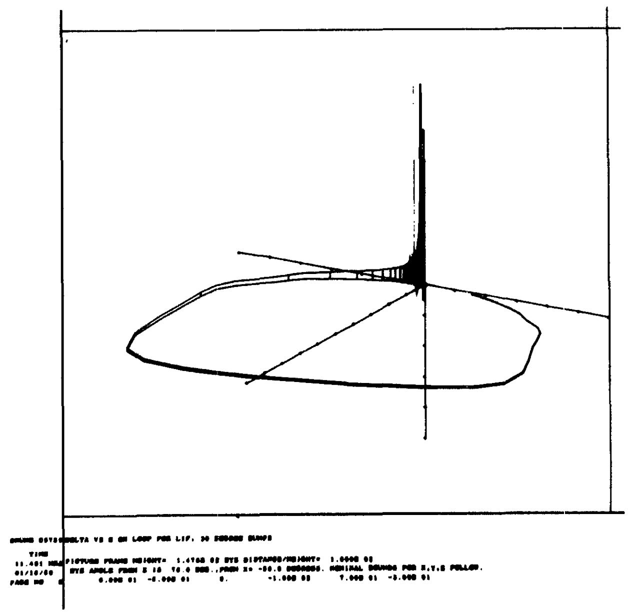
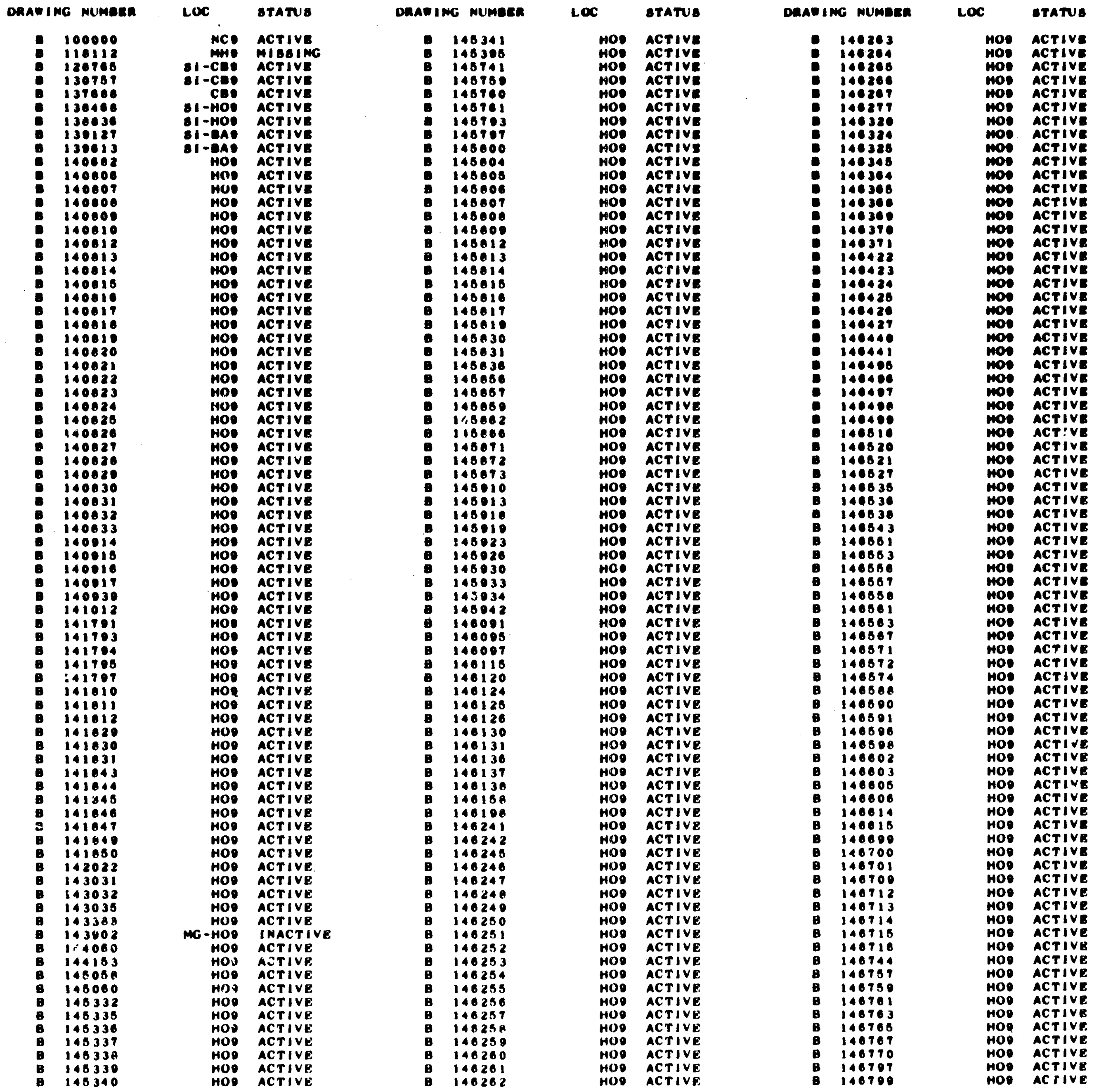


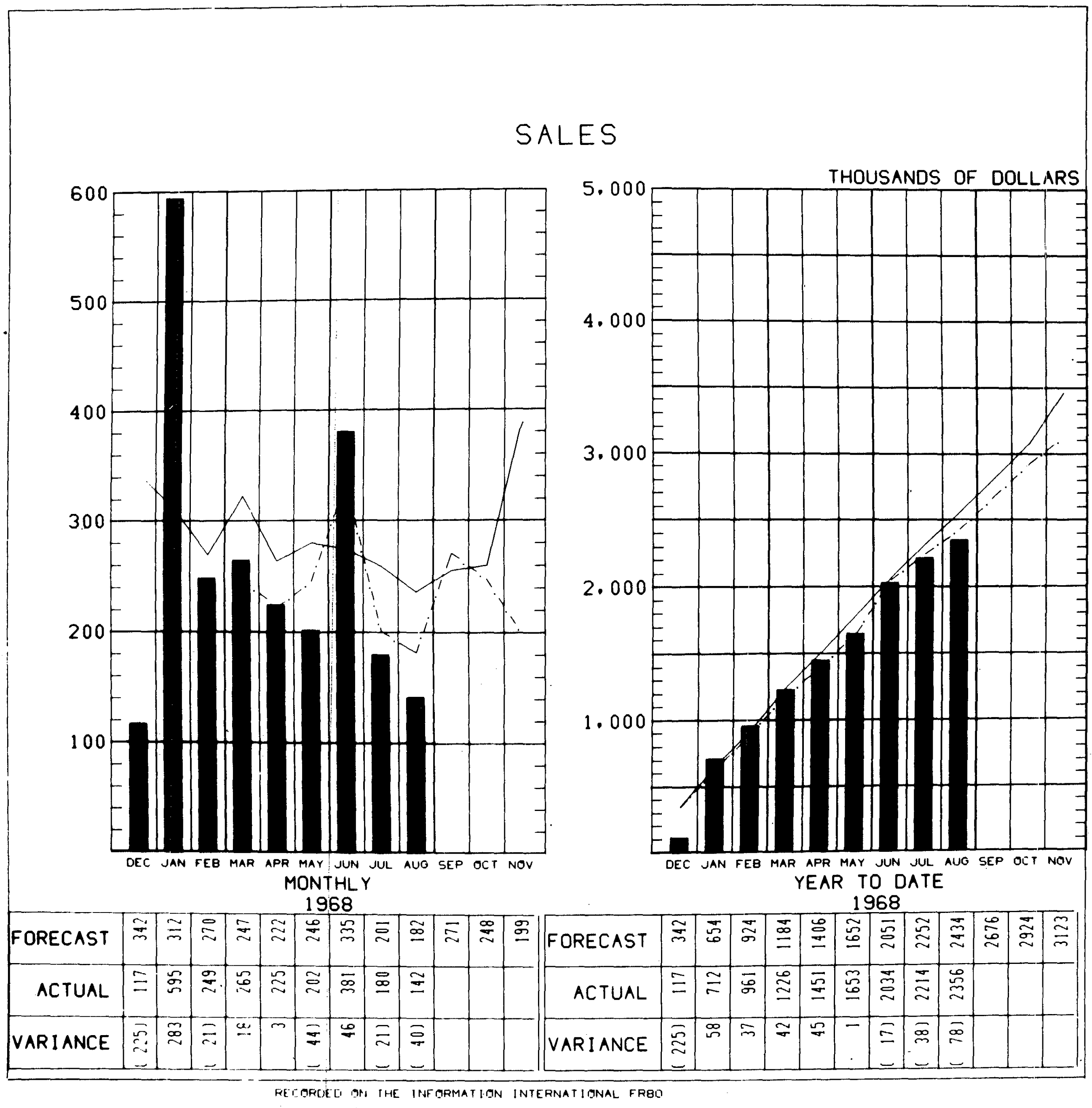
Figure 2—Typical scientific plot



| DRAWING NUMBER | LOC | STATUS | DRAWING NUMBER | LOC | STATUS | DRAWING NUMBER | LOC | STATUS |
|---|---|---|---|---|---|---|---|---|
| B 100000 | NC9 | ACTIVE | B 145341 | HO9 | ACTIVE | B 146263 | HO9 | ACTIVE |
| B 118112 | MH9 | MISSING | B 145395 | HO9 | ACTIVE | B 146264 | HO9 | ACTIVE |
| B 128765 | SI-CB9 | ACTIVE | B 145741 | HO9 | ACTIVE | B 146265 | HO9 | ACTIVE |
| B 130757 | SI-CB9 | ACTIVE | B 145759 | HO9 | ACTIVE | B 146266 | HO9 | ACTIVE |
| B 137668 | CB9 | ACTIVE | B 145760 | HO9 | ACTIVE | B 146267 | HO9 | ACTIVE |
| B 138468 | SI-HO9 | ACTIVE | B 145761 | HO9 | ACTIVE | B 146277 | HO9 | ACTIVE |
| B 138836 | SI-HO9 | ACTIVE | B 145793 | HO9 | ACTIVE | B 146320 | HO9 | ACTIVE |
| B 139127 | SI-BA9 | ACTIVE | B 145797 | HO9 | ACTIVE | B 146324 | HO9 | ACTIVE |
| B 139813 | SI-BA9 | ACTIVE | B 145800 | HO9 | ACTIVE | B 146325 | HO9 | ACTIVE |
| B 140802 | HO9 | ACTIVE | B 145804 | HO9 | ACTIVE | B 146345 | HO9 | ACTIVE |
| B 140806 | HO9 | ACTIVE | B 145805 | HO9 | ACTIVE | B 146364 | HO9 | ACTIVE |
| B 140807 | HO9 | ACTIVE | B 145806 | HO9 | ACTIVE | B 146365 | HO9 | ACTIVE |
| B 140808 | HO9 | ACTIVE | B 145807 | HO9 | ACTIVE | B 146368 | HO9 | ACTIVE |
| B 140809 | HO9 | ACTIVE | B 145808 | HO9 | ACTIVE | B 146369 | HO9 | ACTIVE |
| B 140810 | HO9 | ACTIVE | B 145809 | HO9 | ACTIVE | B 146370 | HO9 | ACTIVE |
| B 140812 | HO9 | ACTIVE | B 145812 | HO9 | ACTIVE | B 146371 | HO9 | ACTIVE |
| B 140813 | HO9 | ACTIVE | B 145813 | HO9 | ACTIVE | B 146422 | HO9 | ACTIVE |
| B 140814 | HO9 | ACTIVE | B 145814 | HO9 | ACTIVE | B 146423 | HO9 | ACTIVE |
| B 140815 | HO9 | ACTIVE | B 145815 | HO9 | ACTIVE | B 146424 | HO9 | ACTIVE |
| B 140816 | HO9 | ACTIVE | B 145816 | HO9 | ACTIVE | B 146425 | HO9 | ACTIVE |
| B 140817 | HO9 | ACTIVE | B 145817 | HO9 | ACTIVE | B 146426 | HO9 | ACTIVE |
| B 140818 | HO9 | ACTIVE | B 145819 | HO9 | ACTIVE | B 146427 | HO9 | ACTIVE |
| B 140819 | HO9 | ACTIVE | B 145830 | HO9 | ACTIVE | B 146440 | HO9 | ACTIVE |
| B 140820 | HO9 | ACTIVE | B 145831 | HO9 | ACTIVE | B 146441 | HO9 | ACTIVE |
| B 140821 | HO9 | ACTIVE | B 145836 | HO9 | ACTIVE | B 146495 | HO9 | ACTIVE |
| B 140822 | HO9 | ACTIVE | B 145856 | HO9 | ACTIVE | B 146496 | HO9 | ACTIVE |
| B 140823 | HO9 | ACTIVE | B 145857 | HO9 | ACTIVE | B 146497 | HO9 | ACTIVE |
| B 140824 | HO9 | ACTIVE | B 145859 | HO9 | ACTIVE | B 146498 | HO9 | ACTIVE |
| B 140825 | HO9 | ACTIVE | B 145862 | HO9 | ACTIVE | B 146499 | HO9 | ACTIVE |
| B 140826 | HO9 | ACTIVE | B 145866 | HO9 | ACTIVE | B 146516 | HO9 | ACTIVE |
| B 140827 | HO9 | ACTIVE | B 145871 | HO9 | ACTIVE | B 146520 | HO9 | ACTIVE |
| B 140828 | HO9 | ACTIVE | B 145872 | HO9 | ACTIVE | B 146521 | HO9 | ACTIVE |
| B 140829 | HO9 | ACTIVE | B 145873 | HO9 | ACTIVE | B 146527 | HO9 | ACTIVE |
| B 140830 | HO9 | ACTIVE | B 145910 | HO9 | ACTIVE | B 146535 | HO9 | ACTIVE |
| B 140831 | HO9 | ACTIVE | B 145913 | HO9 | ACTIVE | B 146536 | HO9 | ACTIVE |
| B 140832 | HO9 | ACTIVE | B 145918 | HO9 | ACTIVE | B 146538 | HO9 | ACTIVE |
| B 140833 | HO9 | ACTIVE | B 145919 | HO9 | ACTIVE | B 146543 | HO9 | ACTIVE |
| B 140914 | HO9 | ACTIVE | B 145923 | HO9 | ACTIVE | B 146551 | HO9 | ACTIVE |
| B 140915 | HO9 | ACTIVE | B 145926 | HO9 | ACTIVE | B 146553 | HO9 | ACTIVE |
| B 140916 | HO9 | ACTIVE | B 145930 | HO9 | ACTIVE | B 146556 | HO9 | ACTIVE |
| B 140917 | HO9 | ACTIVE | B 145933 | HO9 | ACTIVE | B 146557 | HO9 | ACTIVE |
| B 140939 | HO9 | ACTIVE | B 145934 | HO9 | ACTIVE | B 146558 | HO9 | ACTIVE |
| B 141012 | HO9 | ACTIVE | B 145942 | HO9 | ACTIVE | B 146561 | HO9 | ACTIVE |
| B 141791 | HO9 | ACTIVE | B 146091 | HO9 | ACTIVE | B 146563 | HO9 | ACTIVE |
| B 141793 | HO9 | ACTIVE | B 146095 | HO9 | ACTIVE | B 146567 | HO9 | ACTIVE |
| B 141794 | HO9 | ACTIVE | B 146097 | HO9 | ACTIVE | B 146571 | HO9 | ACTIVE |
| B 141795 | HO9 | ACTIVE | B 146115 | HO9 | ACTIVE | B 146572 | HO9 | ACTIVE |
| B 141797 | HO9 | ACTIVE | B 146120 | HO9 | ACTIVE | B 146574 | HO9 | ACTIVE |
| B 141810 | HO9 | ACTIVE | B 146124 | HO9 | ACTIVE | B 146588 | HO9 | ACTIVE |
| B 141811 | HO9 | ACTIVE | B 146125 | HO9 | ACTIVE | B 146590 | HO9 | ACTIVE |
| B 141812 | HO9 | ACTIVE | B 146126 | HO9 | ACTIVE | B 146591 | HO9 | ACTIVE |
| B 141829 | HO9 | ACTIVE | B 146130 | HO9 | ACTIVE | B 146596 | HO9 | ACTIVE |
| B 141830 | HO9 | ACTIVE | B 146131 | HO9 | ACTIVE | B 146598 | HO9 | ACTIVE |
| B 141831 | HO9 | ACTIVE | B 146136 | HO9 | ACTIVE | B 146602 | HO9 | ACTIVE |
| B 141843 | HO9 | ACTIVE | B 146137 | HO9 | ACTIVE | B 146603 | HO9 | ACTIVE |
| B 141844 | HO9 | ACTIVE | B 146138 | HO9 | ACTIVE | B 146605 | HO9 | ACTIVE |
| B 141845 | HO9 | ACTIVE | B 146158 | HO9 | ACTIVE | B 146606 | HO9 | ACTIVE |
| B 141846 | HO9 | ACTIVE | B 146198 | HO9 | ACTIVE | B 146614 | HO9 | ACTIVE |
| B 141847 | HO9 | ACTIVE | B 146241 | HO9 | ACTIVE | B 146615 | HO9 | ACTIVE |
| B 141849 | HO9 | ACTIVE | B 146242 | HO9 | ACTIVE | B 146699 | HO9 | ACTIVE |
| B 141850 | HO9 | ACTIVE | B 146245 | HO9 | ACTIVE | B 146700 | HO9 | ACTIVE |
| B 142022 | HO9 | ACTIVE | B 146246 | HO9 | ACTIVE | B 146701 | HO9 | ACTIVE |
| B 143031 | HO9 | ACTIVE | B 146247 | HO9 | ACTIVE | B 146709 | HO9 | ACTIVE |
| B 143032 | HO9 | ACTIVE | B 146248 | HO9 | ACTIVE | B 146712 | HO9 | ACTIVE |
| B 143035 | HO9 | ACTIVE | B 146249 | HO9 | ACTIVE | B 146713 | HO9 | ACTIVE |
| B 143388 | HO9 | ACTIVE | B 146250 | HO9 | ACTIVE | B 146714 | HO9 | ACTIVE |
| B 143902 | MG-HO9 | INACTIVE | B 146251 | HO9 | ACTIVE | B 146715 | HO9 | ACTIVE |
| B 144060 | HO9 | ACTIVE | B 146252 | HO9 | ACTIVE | B 146716 | HO9 | ACTIVE |
| B 144153 | HO9 | ACTIVE | B 146253 | HO9 | ACTIVE | B 146744 | HO9 | ACTIVE |
| B 145058 | HO9 | ACTIVE | B 146254 | HO9 | ACTIVE | B 146757 | HO9 | ACTIVE |
| B 145060 | HO9 | ACTIVE | B 146255 | HO9 | ACTIVE | B 146759 | HO9 | ACTIVE |
| B 145332 | HO9 | ACTIVE | B 146256 | HO9 | ACTIVE | B 146761 | HO9 | ACTIVE |
| B 145335 | HO9 | ACTIVE | B 146257 | HO9 | ACTIVE | B 146763 | HO9 | ACTIVE |
| B 145336 | HO9 | ACTIVE | B 146258 | HO9 | ACTIVE | B 146765 | HO9 | ACTIVE |
| B 145337 | HO9 | ACTIVE | B 146259 | HO9 | ACTIVE | B 146767 | HO9 | ACTIVE |
| B 145338 | HO9 | ACTIVE | B 146260 | HO9 | ACTIVE | B 146770 | HO9 | ACTIVE |
| B 145339 | HO9 | ACTIVE | B 146261 | HO9 | ACTIVE | B 146797 | HO9 | ACTIVE |
| B 145340 | HO9 | ACTIVE | B 146262 | HO9 | ACTIVE | B 146799 | HO9 | ACTIVE |

Figure 3—Typical business information-alphanumerics

These non-scientific (business) applications prompted the development of special COM devices which are designed for high speed recording of alphanumeric computer output. These units record the same type of information as impact printers only they are much faster and the information is placed on microfilm instead of paper. Figure 3 shows an example of this type of information. Thousands of computers in use today do not yield full capacity. The computer systems are slowed down by their output devices, the impact printers, which produce too much paper. The mountains of printout they produce are smothering the very efficiencies for which computers were designed. These thousands of computers do not put vital information into the hands of the right people in the right places in time for the right decisions. These new business



| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FORECAST | 342 | 312 | 270 | 247 | 222 | 246 | 335 | 201 | 182 | 271 | 248 | 199 |
| ACTUAL | 117 | 595 | 249 | 265 | 225 | 202 | 381 | 180 | 142 | | | |
| VARIANCE | ( 225) | 283 | ( 21) | 18 | 3 | ( 44) | 46 | ( 21) | ( 40) | | | |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FORECAST | 342 | 654 | 924 | 1184 | 1406 | 1652 | 2051 | 2252 | 2434 | 2676 | 2924 | 3123 |
| ACTUAL | 117 | 712 | 961 | 1226 | 1451 | 1653 | 2034 | 2214 | 2356 | | | |
| VARIANCE | ( 225) | 58 | 37 | 42 | 45 | 1 | ( 17) | ( 38) | ( 78) | | | |

Figure 4—Typical business report-graphics

COM recorders can solve this problem. The problem is solved by the following advantages the COM system has over the impact printing system:

1. Printing at computer tape speeds.
2. Forms printed with data simultaneously.
3. Retrieval coding placed on records as it is created.
4. Smaller records storage.
5. Reduced cost of supplies and material.
6. Weight of information significantly reduced.
7. Microfilm doesn't have to be decollated, burst or bound.

The third type of COM is the graphic arts printer. This is an electronic composition system. This type of recorder can produce alphanumerics and graphics with graphic arts quality at data processing speeds.

The evolution of the COM is quite interesting, it began with the scientific device being used for plotting technical data in graphic form. Now it is being used extensively by business for alphanumerics as a replacement for impact printers. I predict that business management will quickly realize that they too would have great advantages from the scientific type of system and have their business information plotted and presented in graphic form instead of as alphanumerics or having a draftsman manually prepare charts from alphanumeric data. Figure 4 shows a business report produced by a scientific COM recorder.

*Technology*

### Speed

The most obvious technological advantage of a COM is the speed at which computer information is translated into human readable form on microfilm. It is difficult to visualize or appreciate this speed. I, therefore, present these comparisons:

| 5,000 electric typewriters | = | 30 impact printers | = | 1 COM recorder |
|---|---|---|---|---|

Looking at it another way:

| | Characters per sec. | Lines per min. | Lines per hr. | Pages per hr. |
|---|---|---|---|---|
| Typewriter | 15 | 7 | 400 | 6 |
| Impact Printer | 2420 | 1100 | 66,000 | 1031 |
| COM recorder | 70,000 | 32,000 | 1,900,000 | 30,000 |

### Cathode ray tube (CRT) systems

The Computer Output Microfilmer, as the name implies, produces computer generated microfilm records with no intervening paper copy. This is achieved by converting the computer digital signals to voltages which are applied to a cathode ray tube. (Another method, electron beam recording, will be described later.) This conversion process results in the information being displayed on the cathode ray tube screen in human understandable form. The microfilm record is produced by photographing the information displayed on the cathode ray tube. The basic nature of this process is illustrated in Figure 5.

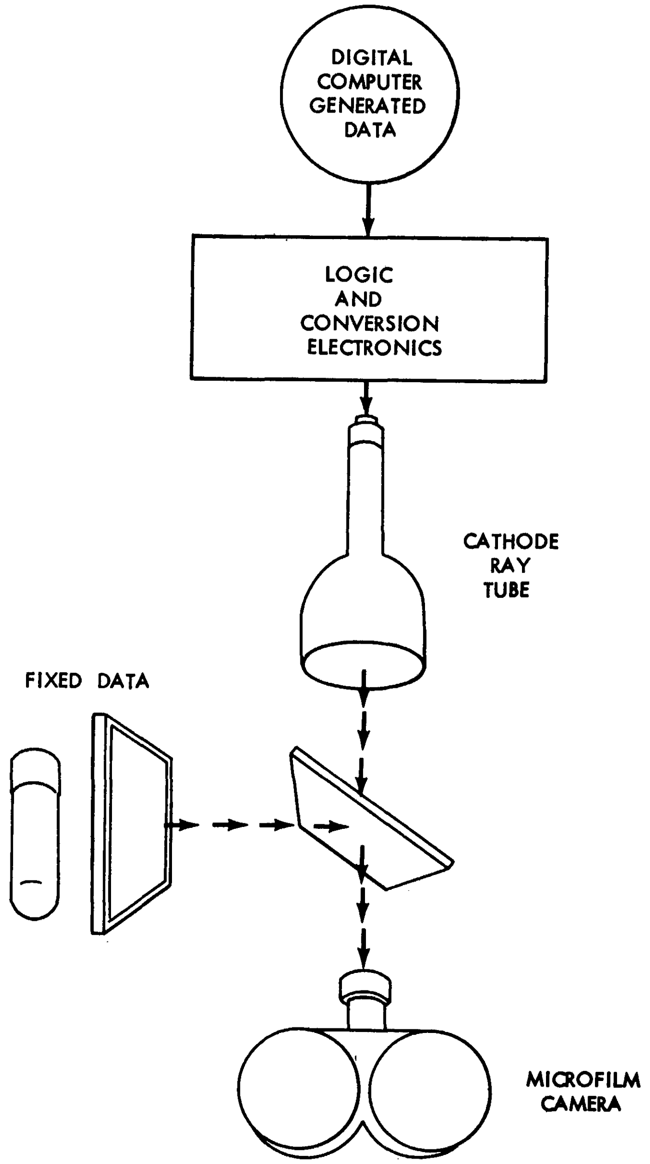

DIGITAL COMPUTER GENERATED DATA
LOGIC AND CONVERSION ELECTRONICS
CATHODE RAY TUBE
FIXED DATA
MICROFILM CAMERA

Figure 5—CRT microfilm recording

### Electron beam recording (EBR) systems

The second method of recording directly on microfilm uses an electron beam, see Figure 6. Using the stroke method, much like that of a pencil writing on paper, the electron beam writes a latent image directly on dry-silver microfilm. The electron beam originates at the cathode of the electron gun, located on the top of a sealed housing. Electrostatic plates and electromagnetic yokes, or magnetic lenses, deflect the beam to form characters and position them on the microfilm frame. The housing is similar in principle to a

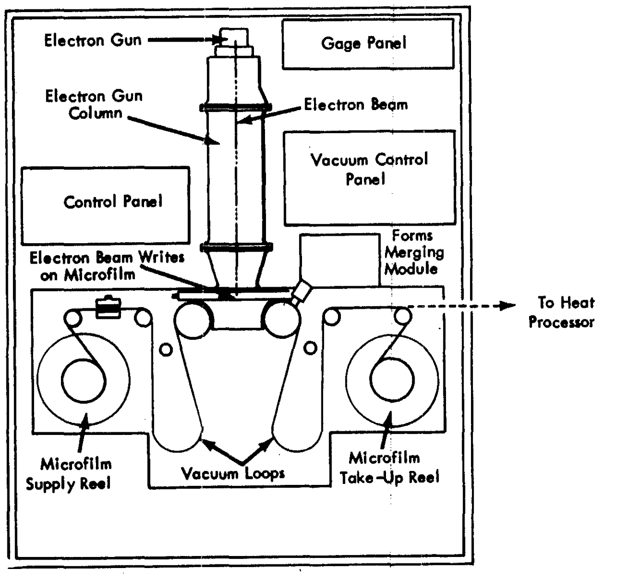


Figure 6—Electron beam recording on microfilm

cathode ray tube except in place of a phosphor screen it has a small aperture through which the beam passes to write directly on the microfilm. Vacuum pumps reduce the air pressure within the housing to a level low enough to facilitate generation and precise control of the beam. Because the beam has practically no inertia, it can be deflected, or modulated, rapidly enough to keep pace with the data transfer rates of the tape drive.

### Character generation

There are several methods of creating characters for COM recording. Stromberg-Datagraphix has developed a special cathode ray tube called a Charactron®* Shaped Beam Tube. The Charactron tube creates an image by directing an electronic beam through individual characters cut in a matrix—a thin precise disc with alphanumeric and symbolic characters etched through it. This matrix is located within the neck of the tube. This method extrudes the beam into the shape of the character being printed. This has the effect of stenciling each character onto the tube face.

Another method of creating characters is by the use of a "stroke" generator. In this type of system a spot is deflected to trace the shape of the character desired. The voltages necessary to deflect the spot are generated by sweep generators, one for X deflection and one for Y deflection. Instructions for the characters are stored in memory. About 16 strokes are used on an average per character.

Characters can also be created by point plotting. This method is generally used for special symbols or type faces.

### Line generation

Line generation in scientific COM recorders is done by the use of a "line" or "vector" generator. This is known as a vector stroke generator and is capable of drawing vectors. Line width, vector direction and intensity levels are all generally programmable.

### Forms overlay

Forms overlay features are provided on most units. The forms overlay feature provides the capability of superimposing predetermined, fixed forms with the generated image. Forms are interchangeable by an operator or on some units may be called in by program. These forms may contain maps, company logos, charts and graphs such as the one in Figure 7.

### Retrieval coding

COM recorders can generate retrieval codes and patterns for each or selected frames of information. The following coding systems are usually standard features: Codeline, Image Blip (Image Count) and Miracode. These indexing identifiers are recorded on film simultaneously with the data. This feature is the key to push button easy retrieval of information on microfilm.

### Films

There are two recording films in use today in the COM field. Almost all CRT systems use Kodak (Recordak) Dacomatic* film, types 5461 and 7461. The EBR systems use 3M Computer Film, type 761 (dry-silver). The Dacomatic film is available in the following sizes:

a. 105mm nonperforated
b. 35mm with perforations
c. 16mm with perforations
d. 16mm nonperforated

The 3M dry-silver film is available in 16mm non-perforated form.

The 105mm film is used in the business type COM and the film is cut and used as microfiche. The 35mm

* Charactron is a trademark of Stromberg-Datagraphix, Inc.

* Dacomatic is a trademark of Eastman Kodak.

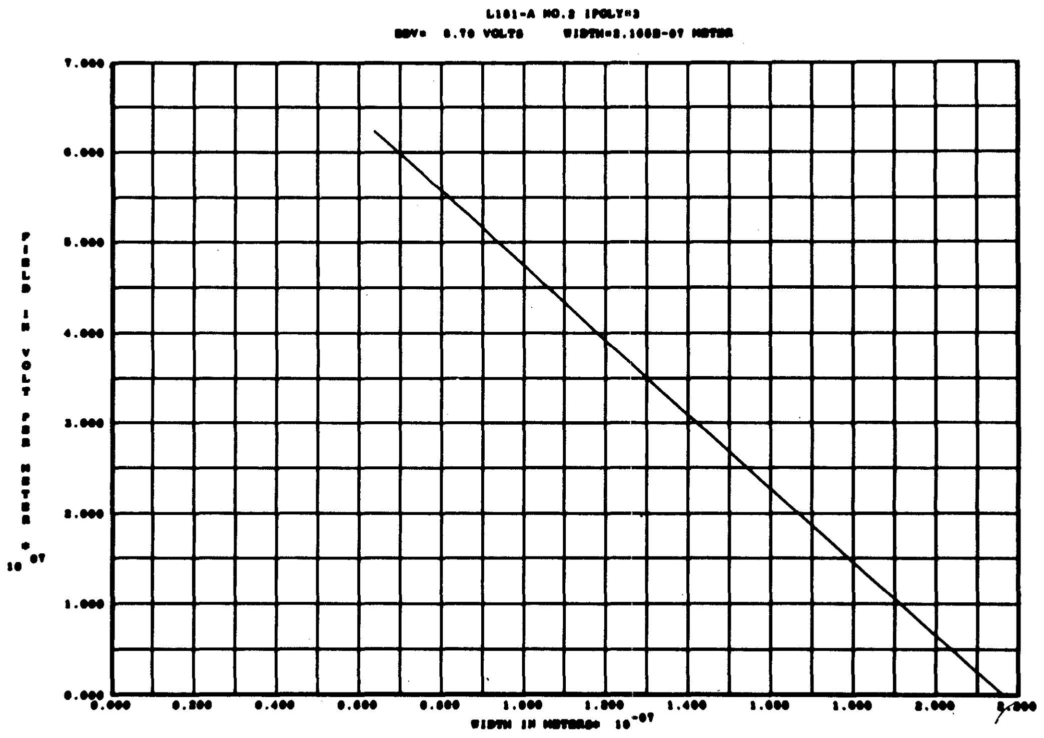


Figure 7—Forms overlay

film is generally used for scientific work (graphics). The 16mm nonperforated film is used for almost all business applications. The perforated 16mm film is only used for special high precision applications.

*Users and applications*

### Scientific

A few of the scientific applications are: circuits, printed wiring board masters, thin film masks, animated movies, graphs and charts. See Figures 1, 2 and 3. The following are some of the organizations using scientific COM recorders:

North American Aviation
NASA
Collins Radio
Bell Telephone Laboratories
Lawrence Radiation Laboratories
MIT Lincoln Laboratories

### Business

Business applications include all types of listing; account reports, management reports and anything that might have been produced by a computer and impact printer. The following are some of the organizations using COM's in business applications:

Sears Roebuck & Company
J. C. Penney Company
Social Security Administration
Equitable Life Assurance Society
Bureau of the Census
International Harvester Company

### Systems service centers

At the present time there are over 40 systems service companies operating COM service centers in the following cities in the United States:

*California*
- Canago Park
- Culver City
- El Segundo
- Glendale
- Los Angeles
- San Francisco
- Stockton

*Colorado*
- Bolder

Denver
*Connecticut*
Hartford
Westport
*Florida*
Miami
*Georgia*
Atlanta
*Illinois*
Chicago
*Indiana*
Indianapolis
*Louisiana*
New Orleans
*Maryland*
Baltimore
College Park
*Massachusetts*
Boston
Springfield
Wilmington
*Michigan*
Dearborn
Detroit
*Missouri*
St. Louis
*New Jersey*
Cherry Hill
Dayton
*New York*
Binghamton
Buffalo
New York City
Rochester
Spring Valley
White Plains
*North Carolina*
Winston-Salem
*Ohio*
Columbus
Cleveland
Dayton
*Pennsylvania*
Philadelphia
Pittsburgh
*Texas*
Austin
Dallas
Houston
*Utah*
Salt Lake City
*Virginia*
Arlington
*Washington, D. C.*
*Wisconsin*
Brookfield

## COM recorders

At the writing of this paper the followng companies were marketing COM units:

a. AMETEK/Straza (Scientific)
b. Beta Instrument (Scientific)
c. California Computer Products (Scientific)
d. Canon (Business)
e. Computer Micro–Data Systems (Scientific)
f. Computer Industries (Scientific & Business)
g. Control Data (Scientific)
h. Eastman Kodak (Business)
i. Information International (Scientific)
j. 3M (Business)
k. RCA (Graphic Arts)
l. Scan Graphics (Scientific)
m. Singer-Link (Scientific)
n. Stromberg-Datagraphix (Scientific & Business)

### *Total COM systems*

As can be seen in Figure 8 there is very little difference between photographing a CRT or a paper document. In selecting the film for recording from a CRT it should be matched to the phosphor of the tube in sensitivity. The polarity of the image on a CRT is negative (light lines on a dark background) and on paper it is usually positive. Therefore, with normal film processing the image of the CRT will be reversed and appear on film as a positive and a microfilm of a positive paper document will appear as a negative. Since most users of microfilm prefer to use negative images in readers and for making hardcopy it is necessary to obtain a negative image of the COM film, this is done one of two ways. At the time of processing the recording film is flashed and developed in a special movie processor which provides a negative image on film from the negative CRT image. The second method of obtaining negative film images is to make a second generation duplicate on Kalvar or silver film which will reverse the polarity again and therefore from a negative CRT image we get, with normal processing, a positive first generation recording film and then a negative second generation duplicate.

Figure 9 depicts the various systems used for COM operations. In scientific applications the film is most often put in aperture cards or used as short strips or on reels. In most business applications the film is used in roll from in cartridges. There are a few systems where the film is cut and pasted up to make a master microfiche. A recent development, the 105mm film head for a COM provides microfiche directly and therefore eliminates much of the manual labor in producing microfiche. In most business applications film duplicates are required to disseminate the information to many users. In all systems, readers, reader-printers, retrieval devices and enlarger-printers are needed by the end users of microfilm. Additional information on these items can be obtained from the National Microfilm Association's "Guide to Microreproduction Equipment" now in its fourth edition.

There are six generally used methods of making copies of computer generated reports. Figure 10 provides a cost comparison of a 100 page report. As can be seen on the graph, distribution of microfilm duplicates is the lowest cost method at any quantity of copies.

*The COM market & NMA survey*

At the end of 1968 there were about 300 COM recorders in use. Of this number about 60 units were being operated by systems service companies.

There have been many forecasts made of the COM field with as many conclusions as studies. Figure 11 shows the range of these forecasts, which is that by 1975 there will be between six and 12,000 recorders in operation. The cost of a COM is $60,000 to $300,000 with the average being about $100,000. This average rental is in the order of $40,000 per year.

In the Spring of this year the National Microfilm Association made a survey of all its over 3,000 members. with regard to the use of Computer Output Microfilm. The following are some of the statistics obtained and *my* comments:

1. Questionnaires were returned by 24 percent of those queried.

*Comment*: 24 percent is considered an excellent return on a direct mail survey. NMA

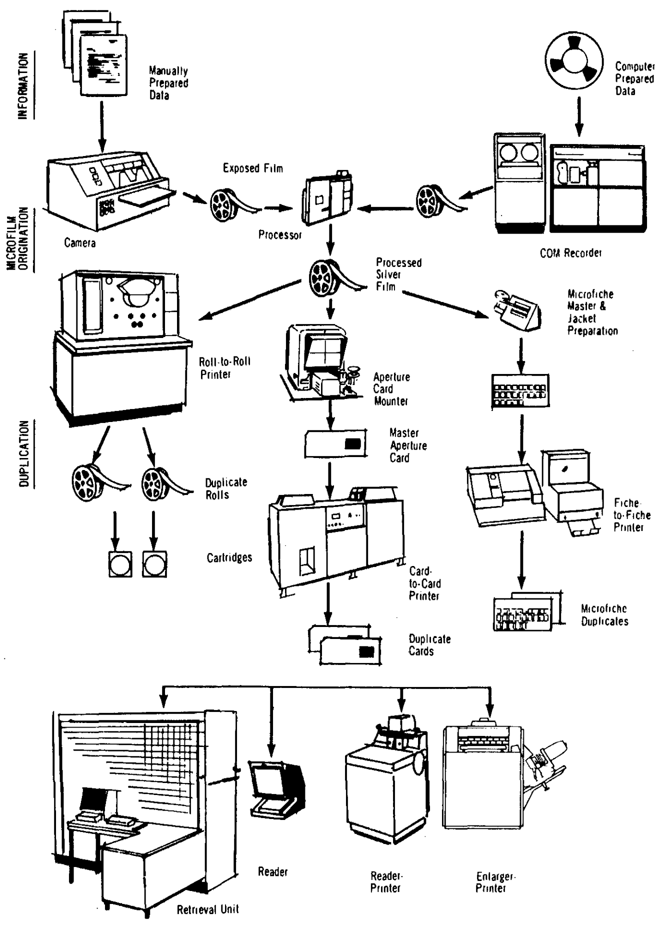


Figure 9—Micrographic information systems

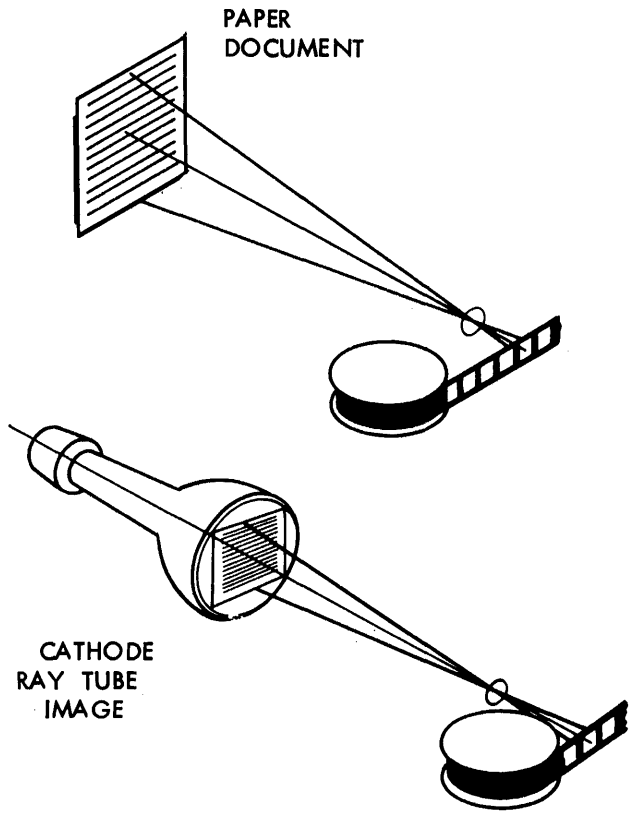


Figure 8—Microfilming systems

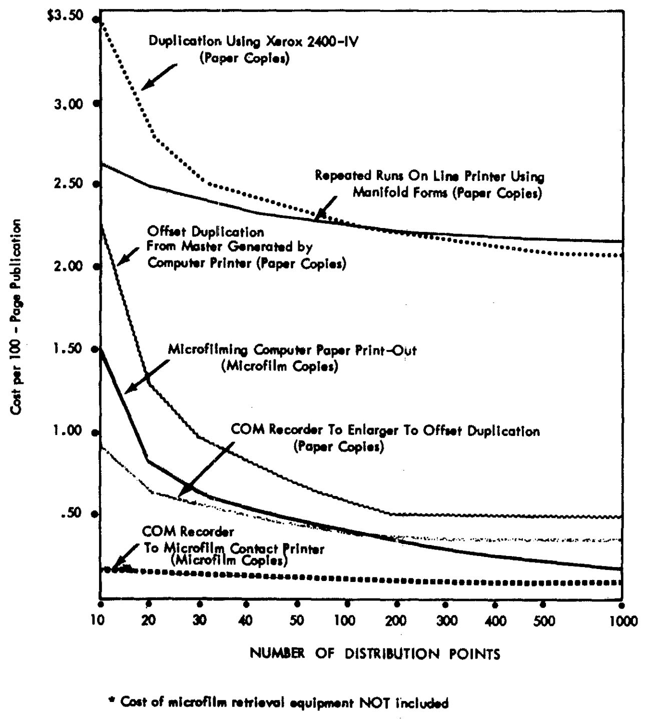


Figure 10—Comparative costs of creating copies of computer generated output

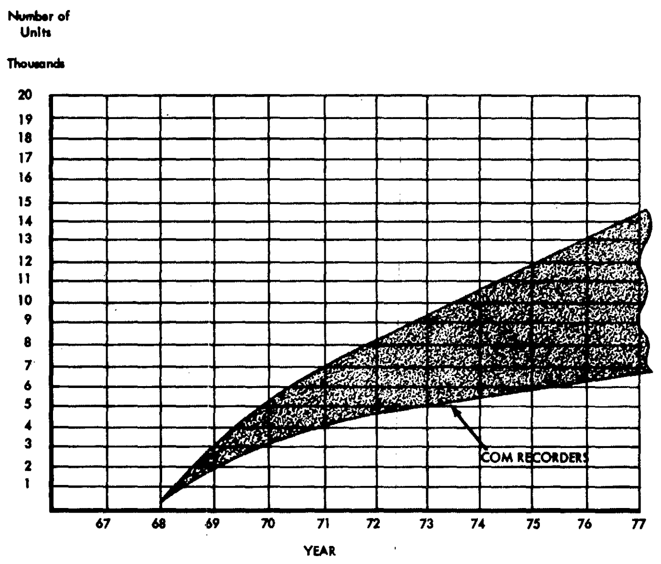


Figure 11—COM recorder unit placement forecast

members are interested in the COM field.

*NOTE*: SYSTEMS SERVICE COMPANIES HAVE *NOT* BEEN INCLUDED IN ANY OF THE FOLLOWING STATISTICS.

2. Questionnaires were returned by 74 organizations indicating they *now* have a COM recorder(s), 29 were scientific and 59 business units.

   *Comment*: 33 percent of COM's are scientific type today 67 percent of COM's are business type today

3. 105 organizations indicated they would obtain their first COM in the next two years. 28 organizations indicated they would obtain an additional COM in the next two years and 55 organizations indicated the use a COM was under study.

   *Comment*: By the end of 1970 there will probably be more than 1,000 COM's in use.

4. Positive versus negative *original* recording film:
   56 using positive (normal processing)
   29 using negative (flash reversal processing)
   A few organizations use both positive and negative

   *Comment*: Even though it requires special processing equipment to obtain a negative image on the original film it is being done, there must be a need.

5. The following film processors are being used in COM systems:

| | |
|---|---|
| Fulton | 10 users |
| Kodak | 18 users |
| Remington Unipro | 3 users |
| Stromberg | 6 users |
| Other | 35 users |

6. The following is the quantity of original recording film being used per month by 48 respondents who gave figures:

| | |
|---|---|
| 16mm perforated | 80,000 feet |
| 16mm nonperforated | 852,000 feet |
| 35mm perforated | 62,000 feet |
| 35mm nonperforated | 31,000 feet |
| 105mm nonperforated | 6,000 feet |

   *Comment*: The following is an estimate of the quantity of recording film all COM's are currently using per month.

| | |
|---|---|
| 16mm perforated | 400,000 feet |

| | |
|---|---|
| 16mm nonperforated | 4,200,000 feet |
| 35mm perforated | 300,000 feet |
| 35mm nonperforated | 150,000 feet |

7. 57 of the 74 users duplicate their film.

   23 Diazo
   29 Kalvar
   22 Silver

   Some used more than one process.

   *Comment*: Convenience and turnaround time are most important.

8. Duplicating film being used per month by 40 COM systems reporting:

   16mm—3,800,000 feet
   35mm—35,000 feet
   105 x 148mm—34,000 fiche
   3–1/4″ x 7–3/8″—270,000 fiche
   6″ x 8″—34,000 fiche
   Aperture cards—367,000 cards

   *Comment*: The following is an estimate of the quantity of duplicating film being used per month by all COM systems:

   16mm—22,800,000 feet
   35mm—200,000 feet
   Aperture cards—2,200,000 cards
   Microfiche (various sizes)—2,000,000 fiche

9. Microforms being used in COM systems reporting:

   Roll Film (including cartridges)—56 users
   Microfiche—21 users
   Jackets—13 users
   Aperture cards—13 users

   *Comment*: The following are the percentages of COM systems using each microform:

   Roll film (including cartridges)—55 percent
   Microfiche—21 percent
   Jackets—12 percent
   Aperture cards—12 percent

10. For those using roll film and cartridges the following indexing systems are in use:

   Miracode*—10 users
   Image Blip (Image Count)—24 users
   Code Line—7 users
   Flash—9 users
   Other—12 users

   *Comment*: Most COM systems are now using the Image Blip (Image Count) system of retrieval.

11. Regarding a question on the use of hardcopy the following responses were received:

   Never used—5 users
   Seldom used—20 users
   Frequently used—34 users
   Always used—6 users

   *Comment*: Hardcopy is required, but on a selected basis.

12. For 35 respondents, 844,000 pages of hardcopy are produced each month.

   *Comment*: The average COM system produces 24,000 pages of hardcopy per month.

*Standards*

In February of 1968 the National Microfilm Association (NMA) established a committee to investigate and recommend standards for microfilm produced by COM recorders. This committee has members from most of the COM manufacturers, several COM systems service companies and many users in government and industry. There are three sub-committees each with a mission as follows:

Format, Quality and Glossary.

The National Microfilm Association is attempting to coordinate the activities of this new microfilm application by considering standards, reporting of many specific applications in its Journal and having COM exhibits at its annual convention.

For additional information on the COM field, write to the National Microfilm Association, P.O. Box 386, 250 Prince George Street, Annapolis, Maryland 21404.

* Miracode is a trademark of Eastman Kodak.

# The microfilm page printer—Software considerations *

*by* S. A. BROWN

*Datalogics, Inc.*
Chicago, Illinois

## INTRODUCTION

Magnetic tape microfilm recorders have been available in the market place for the past several years. It has been only within the last eight months or so that a general awareness of these devices has developed. Trade magazines and journals are now carrying feature articles describing computer based, microfilm information systems. Investment houses are releasing surveys and market evaluations of this area. Talks on the economics and human engineering aspects of this approach are being presented at many technical conferences. Little, however, has been said about programming considerations for the preparation of the specially formatted magnetic tape required for the operation of these devices. The purpose of this paper, then, is to examine the flexibilities and capabilities of magnetic tape microfilming as viewed by the programmer, to discuss the software problems that he faces when attempting to use such a device and to describe several generalized solutions to these problems.

### *The machine*

A typical such device is the Series F Electron Beam Recorder manufactured by 3M Company. It is a microfilm page printer with an extended graphic set. The page area is a 132×64 character array organized as 64 lines each containing maximally 132 characters. Data to be printed reside as line images on magnetic tape. Each line is represented as a character string prefaced by one or two characters of coordinate data and terminated by delimiter character. The coordinate characters specify the position of the line on the page. This specification may be in absolute, in terms of a specific character in the page array, or relative to the last printed line.

A page printer differs from a conventional impact line printer in allowing the line to be placed randomly on the page rather than in ascending line sequence. It is possible to skip from the bottom of a page to the top as easily as from the top to the bottom. This is illustrated in Figure 1.

This may be contrasted to an impact printer which can only advance.

The flexibilities of a page printer can best be appreciated by considering the problem of printing a report containing, say three vertical columns. To print such a report on a conventional line printer would require buffering an entire page in core or at least the first two columns before any data could be printed. In a page printer environment, this restriction is removed.

## Extended character set

Electronic, rather than mechanical generation of the character set provides a wide variety of available graphics. In addition to the standard upper case set, most microfilm printers provide a lower case as well as a bold face. Series F Electron Beam Recorder further includes a large size set.

This graphic variety allows design of highly legible microfilm documents that previously could be obtained only at the expense of typesetting.

* The work described in this paper was sponsored by 3M Company, Computer Graphics, St Paul, Minnesota

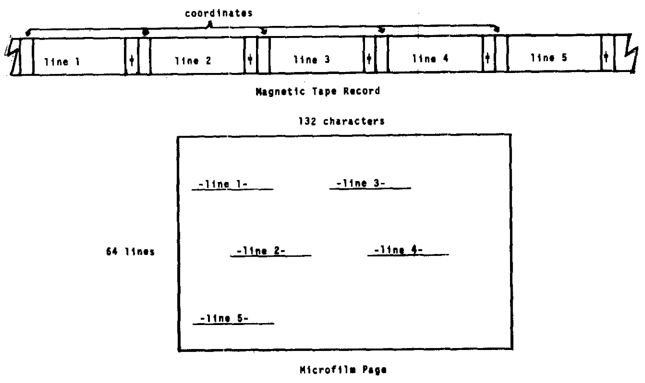


Figure 1—Microfilm page organization

## Forms or line art

This machine provides a means of inserting line drawings or ruled forms with printed headings similar to custom line printer forms. This capability allows insertion of single fixed forms, random retrieval from a library of 30 images and sequential retrieval from a file of 2000 images.

## Applications

The application for this device ranges from that of replacing current impact line printers printing on stock or custom forms to preparation of material that is typeset or types, such as illustrated parts catalogues and directories.

### *Software implications*

Typically, a microfilm printer is used to replace some or all of the functions of a line impact printer. Conceptually this is easy to visualize, line printing is a subset capability of page printing. The user, however, finds himself in one of two situations. Either his program prints directly on-line or formats a tape for off-line printing. The former case obviously implies changes to the application program; the latter implies either program modification or a tape to tape transcription pass. The low unit page cost exhibited by microfilm recorders makes even a nominal tape to tape computer charge relatively expensive. It may amount to 20-40 percent of the total microfilm cost. The apparent alternative is program modification. Typical program conversion takes from two hours to two days, depending on the availability of program source, documentation and test data. Although, the reprogramming time is minimal for a single program, if the universe consists of hundreds, as is normal, a major expenditure of effort is required. Further, in those instances where the user wishes to retain the original program for back-up purposes, he is forced to maintain both programs. The user is confronted with the potential requirement for a large re-programming investment and must weigh this against the economics of a microfilm system.

The user is in a similar position when he requires the extended character set, page printer or forms capabilities. This time, however, he is really modifying his application and can be expected to expend programming effort. He has more than a simple media conversion problem to solve; he is designing a microfilm format that did not previously exist.

In so modifying his application, he has to consider all of the characteristics and idiosyncrasies of the specific microfilm printer he is going to use. These include placement of inter-record gaps and control codes within the text of the microfilm document.

### *Solutions*

In the best of circumstances the user would prefer to see extensions to his operating system and programming languages to support output devices with extended graphic and page printing capabilities. If he desires merely the same output on microfilm that he obtained on hard copy, he should have to change only a peripheral assignment statement in his job deck and execute his program. In the case where the user requires full utilization of the microfilm device's capabilities, he would prefer to resort to new statements in his application languages such as COBOL, PL/I, RPG, etc. These might include facilities for declaring multicolumn output, invoking alternate character sets or specifying insertion of graphics.

3M Company has recognized the need for system software with these capabilities and feels that as the computer microfilm user community grows, operating system and language implementors will include them in future systems. During the initial design phases of the Series F EBR, they asked us to formulate interim solutions for several specific computer systems. We were instructed that these solutions remain valid until such time that microfilm page printers were recognized by operating system implementors as standard peripherals. These solutions can be categorized as either conversion support or new application support.

## Conversion support

Support software has been written for the IBM 360 DOS and O/S operating systems. This has taken the form of extensions to the operating systems. Considerable care has been taken to insure that the change was local and did not disturb the rest of the system.

The DOS extension is a supplement to DOS Logical IOCS and provides object program compatibility with problem programs written in PL/1, COBOL, RPG and Assembly Language. I/O Modules similar in concept to the ones that comprise Logical IOCS were written to interface a printer file definition with a physical magnetic tape drive. This interface routine is responsible for adding the EBR control codes to the print image and forwarding it to the magnetic tape drive. A series of these routines reside, together with standard IBM supplied I/O routines, in the relocatable library. To invoke the extension, the user adds a single link editor control card and re-links his program. The output that normally appeared on a line printer is directed to magnetic tape in a format appropriate to the EBR.

The result of processing this tape on the microfilm page printer is identical in all respects to that previously obtained on the line printer.

A similar extension was provided for IBM's O/S 360 operating system. In this case the user is provided with load module compatibility. Extensions were written for the four QSAM move and locate mode modules. The modules have been modified to examine the volume serial number of the output data set and if the first three characters are "EBR" and the data set is in ASA mode, the file contents are re-formatted before being written on its assigned device. Operationally, the user is required to include only one control card to divert his output from a standard system output writer (SYSOUT) to a magnetic tape in EBR format.

A similar system involving modification of the IBM 1401 Autocoder assembler provides source language compatibility for the EBR. Further object program support is being developed for the CDC 3300, GE 400 and RCA Spectra 70 TDOS operating systems. I personally feel that object program support is extremely important, particularly in this age of proprietary software where source programs may not even be available.

### New application support

Here, the user requires an output format not obtainable on a conventional line printer. He must develop a new program or at least modify a current one. Again, he should be insulated from certain details of the microfilm printer, such as placement of control codes and inter-record gaps. The approach in this case was to provide a general purpose output package written in COBOL. This package, called EBRPACK, provides entry points to select form overlays and character sets, plus additional entries to replace the standard COBOL printer command "WRITE dataname 1 AFTER ADVANCING dataname 2 LINES."

## SUMMARY

Software support for microfilm page printers is necessary and desirable; it must utlimately come from operating system and programming language implementors. In the interim, operating system extensions providing microfilm-line printer interchangeability may readily be prepared. Applications requiring specific features associated only with microfilm page printers may be designed and implemented utilizing output packages written in machine independent languages.

# Computer microfilm—A cost cutting solution to the EDP output bottleneck

*by* JOHN K. KOENEMAN and JOHN R. SCHWANBECK

*Oppenheimer & Company*
New York, New York

## SUMMARY

Although the computer microfilm recorder has received little attention to date, this new output device represents a technological breakthrough which will have a major impact on the computer industry. Installation of a recorder generally results in a tenfold increase in the speed of computer output and a concomitant substantial reduction in CPU time which can result in major data processing and report production cost savings. As an added bonus, a microfilm system is the equal of most electronic time-sharing systems for information storage and retrieval applications. Consequently, we feel that computer microfilm, although little noticed thus far, represents a major industrial and investment concept.

The electromechanical line printer—heretofore the only practical means of obtaining hardcopy rapidly from the computer—has a maximum output rate of only 2,500 characters per second. But, the computer's throughput capability is 25,000 to 100,000 characters per second. Owing to the severe output bottleneck that results from this imbalance of speeds, the bulk of information ingested and produced by computers has, until now, essentially been locked on magnetic tape and not easily available to the computer user. With the advent of the computer microfilm recorder, which can produce output as fast as the computer can process data, this mass of stored information has suddenly become readily available in humanly readable form. One of the most important questions which must therefore be asked is: "How much information is stored on magnetic tape and how badly is it desired by the computer user?" Our field work has consistently shown that an early Xerox type phenomenon exists—user volume rises rapidly to meet capacity.

Because the microfilm recorder eliminates the computer output bottleneck, it also results in a major cost savings. This effect is most readily apparent in the data processing service industry, where a customer can now realize an approximate 40 percent to 50 percent reduction in his monthly service bureau bill if microfilm rather than continuous paper froms is accepted as computer output.

Even greater relative savings can be realized by companies with medium to large-scale in-house data processing departments. Overall, it can be shown that the lowest data processing costs, at all levels of use, are achieved when microfilm recorders are employed to produce alphanumeric or graphic computer reports.

Moreover, acceptance of computer output in film form automatically creates an information storage and retrieval system which is the equal of most electronic systems. Although microfilm has gained a bad reputation because of the poorly designed equipment and improperly processed film which library users have been forced to endure for years, newly introduced microfilm equipment can now easily provide the quality of image and speed of retrieval of the most expensive time-sharing terminals.

In addition to the standard data processing market, there is another separate and distinct market, that of pure information storage and retrieval, for which com-

puter microfilm can compete very effectively because of its low cost. In fact, computer microfilm is frequently referred to as "the poor man's time-sharing." The service bureau charge for processing and producing one page of computer generated microfilm daily for one month is 10 percent to 40 percent that of storing one page of information on magnetic disc for the same time period. When the terminal and communications costs of electronic time-sharing systems are also considered, the cost advantage weighs even more heavily in favor of microfilm. In large measure, this dramatic cost difference is the result of the substantially greater density of data storage which film (1,000,000 bits/sq. cm.) enjoys over magnetic media (1,000 bits/sq. cm.). Thus, although highly optimistic forecasts have been made for the growth of electronic systems for use in information storage and retrieval applications, we feel that fundamental economic considerations strongly suggest that computer generated microfilm, instead, will become the most common (although, obviously, not the sole) method of computer information storage and retrieval.

Several other benefits are derived from the computer microfilm recorder which are normally of peripheral, qut can on occasion be of prime, importance:

- An unlimited number of report copies can be obtained from one computer run with no loss of clarity; by contrast, only four or five truly readable copies can be obtained from a single run when an impact line printer and continuous paper forms are used.
- Owing to its compactness, microfilm essentially eliminates the problems and costs of computer report storage.
- Microfilm permits dramatic reductions in computer report transportation or communications costs.

Computer microfilm is not without certain drawbacks, however. A computer microfilm information system cannot be used in situations where the data base changes rapidly, such as in airline reservations or stock market quotations. It also cannot be employed where user interaction with the data base is desired. Additionally, paper possesses a distinct advantage as data processing output where computer usage is very light, or scientific applications (i.e., high computation—low output) are involved.

In summary, with the development of the computer microfilm recorder, the most efficient processor of information—the computer—has finally been directly linked with the most efficient means of information storage and retrieval—microfilm. User experience to date strongly suggests that very large and potentially vast demand exists for the inexpensive and fast access to computerized information that this combination provides. Indeed there is every indication that computer microfilm could bring about a real information explosion. Certainly all ingredients necessary for such pyrotechnics are present—a sudden quantum jump in the speed of information output, low cost, and ease of use (Exhibit 1). As a consequence, we feel that the computer microfilm service, hardware, and supplies industries will experience impressive growth over the near and intermediate term. Indeed, output of microfilm recorders, which should jump from 100 units in 1968 to about 400 units in 1969, presently is production limited.

*The microfilm recorder substantially reduces data processing and report generation costs for all users*

Although there are considerable variations in volume discounts and prime or off shift machine rates, a 50 percent cost saving is common when a data processing service organization customer changes from paper to microfilm as computer output. Similarly, cost reductions of 40 percent to 70 percent have been documented by heavy in-house computer users even though, in most cases, the availability of computer reports has been substantially increased as well. Although the relative cost savings of the in-house user and the service bureau customer are similar, the source of these savings is not. Whereas essentially all the service bureau cost reduction can be attributed to lower computer time charges, the bulk of in-house economies derives from labor and material savings. On balance, however, it can be shown that the lowest data processing costs are always obtained when a microfilm recorder is employed.

## Service center cost reductions

To obtain 1,000 pages (and three carbon copies) of processed information, a data processing service organization customer presently accepting paper output will incur about one hour of IBM 360/30 machine rental at $65.00 per hour and a materials charge of $30.00 for continuous forms. Thus, total service bureau charges for the processing and production of 1,000 pages of information will total about $95.00 when paper is used as the computer output medium.

If, however, a change to computer microfilm is made, the cost of a similar run drops to about $40.00

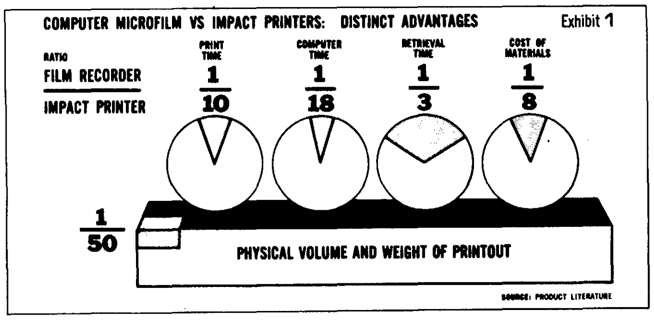

COMPUTER MICROFILM VS IMPACT PRINTERS: DISTINCT ADVANTAGES
Exhibit 1
RATIO
FILM RECORDER
IMPACT PRINTER
PRINT TIME 1/10
COMPUTER TIME 1/18
RETRIEVAL TIME 1/3
COST OF MATERIALS 1/8
1/50
PHYSICAL VOLUME AND WEIGHT OF PRINTOUT
SOURCE: PRODUCT LITERATURE

Exhibit 1—Computer microfilm vs impact printers: Distinct advantages

to $45.00. Because the economics of large, fast computers can be used to advantage when the machine is no longer output bound, most computer microfilm programs are run on an IBM 360/65 or equivalent. Because the time necessary to process 1,000 pages of information on a 360/65 is about 0.2 minutes, total data processing charges at $600 per hour amount to only about $2 or $3. Conversion from magnetic tape to a single microfilm original can be accomplished for about $30.00 (three cents per original page), and the cost of three copies will add an additional $10.00 (3.3 mills per page). Thus, for comparable data processing and report production services, a computer microfilm service bureau will cost only $40.00 to $45.00, in contrast to about $95.00 for a traditional data processing service organization (Exhibit 2).

## In-house cost reductions

In the next exhibit (3), it can be seen that although the installation of a microfilm recorder (SD4360) increases the fixed cost of a data processing installation about $2,000 per month, variable costs for materials are so low that the recorder becomes economically advantageous after 90,000 to 100,000 pages per month of output, or the equivalent of five to six machine hours per day of a relatively small four-tape System 360/30. Thus, an in-house installation operating two shifts can achieve a 25 percent-30 percnet cost reduction through the elimination of machine shift permiums, labor, and materials savings. Extensive Army studies[1] have shown that operating savings of 40 percent to 70 percent can be achieved when three-shift operation or multiple satellite computers with attached line printers are involved.

The magnitude of the demand for computer reports that is presently unsatisfied because such reports are considered uneconomical can perhaps be judged by

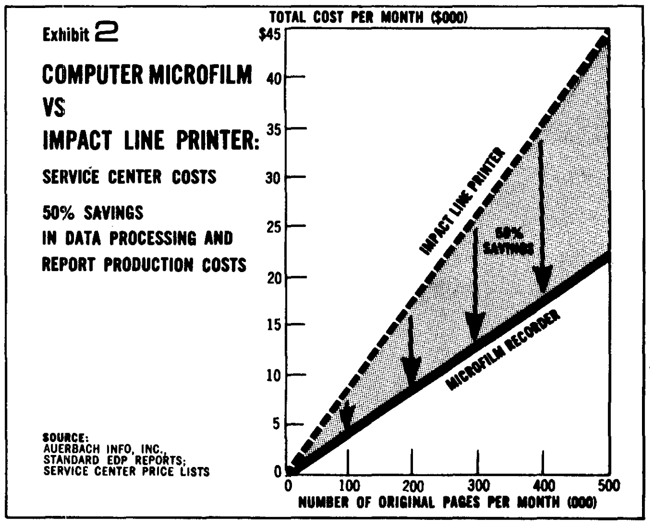


Exhibit 2—Computer microfilm vs impact line printer: Service center costs

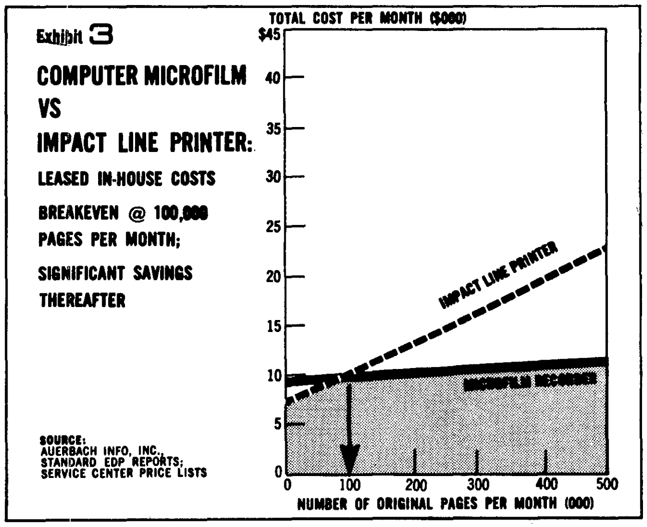


Exhibit 3—Computer microfilm vs impact line printer: Leased in-house costs

noting that if the management of a corporation with as little as $15 million in annual sales desired detailed daily reports on finished parts inventory, accounts receivable, and unfilled orders, almost seven hours of computer time would be consumed in printing out these reports.[2] Incremental costs of about $3,000 to $4,000 per month for materials and possibly $2,000-$2,500 for additional labor would probably thereby be incurred. Thus, although the utility of detailed management reports such as these is probably high,

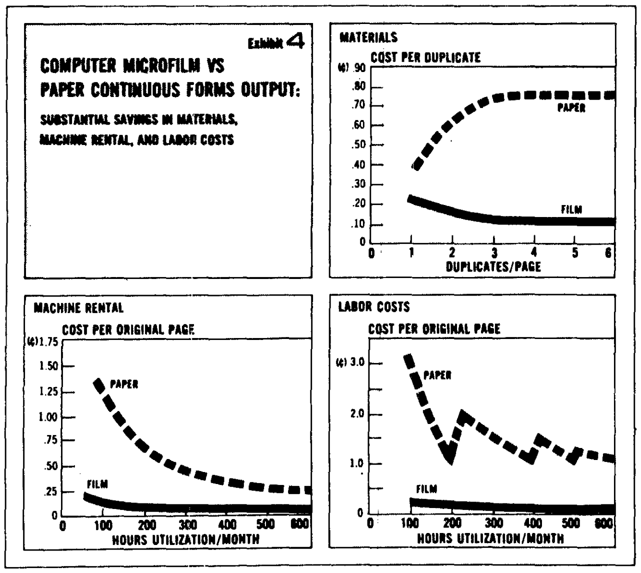


Exhibit 4—Computer microfilm vs paper continuous forms output: Substantial savings in materials, machine rental, and labor costs

we think it likely that the operational difficulties and the extremely high EDP costs necessary to produce such information have led many manufacturing companies to forego such data until now. However, with the installation of a computer microfilm recorder, the same $15 million company described above could produce the same reports at an incremental cost of only $400 to $500 per month for materials and no incremental cost for labor. Thus, the company would then find it feasible to produce these reports. Operating experience to date of computer microfilm recorder owners certainly would point toward such a conclusion.

Moreover, it is important to note that the cost curve of a computer microfilm data processing installation is essentially flat out to very large quantities of output (Exhibit 3). Thus, the corporate manager would now be able to obtain additional detailed reports almost instantaneously at virtually no incremental cost.

Experience to date indicates that most managements will quickly begin to utilize the full capacity of a newly installed recorder.

For example, in one case, a large insurance company installed a microfilm recorder in May 1967. Although the equipment operated only five hours per week when first installed, after approximately one year, utilization had increased tenfold to 50 hours per week. In another case, a manufacturing concern which began using prototype computer microfilm equipment in 1967 had increased its film consumption to 20 million feet per year (400 million pages) by 1967 and reached 38 million feet (760 million pages) in 1968.

The substantial savings in consumable materials costs, labor costs, and machine rental are, of course, the three major cost elements considered in calculating operational savings (Exhibit 4).

Additionally, however, considerable savings in computer report shipping and storage costs can frequently be realized, although these expense elements have not been included in our calculations (Exhibit 5).

**COMPUTER MICROFILM VS PAPER CONTINUOUS FORMS OUTPUT:** Exhibit 5

**SUBSTANTIAL SAVINGS IN STORAGE AND SHIPPING COSTS . . .**

| 5,000 PAGES | VOLUME | APPROXIMATE ANNUAL STORAGE COST |
|---|---|---|
| MICROFILM<br>PAPER | 0.10 Ft$^3$<br>4.50 Ft$^3$ | $0.05<br>$4.15 |
| 5,000 PAGES | WEIGHT | APPROXIMATE FIRST CLASS MAIL COST |
| MICROFILM<br>PAPER | 3.0 Lbs.<br>150.0 Lbs. | 0.08 ¢ / mile<br>4.05 ¢ / mile |

SOURCES: KISH, JOSEPH L. & MORRIS, JAMES – MICROFILM IN BUSINESS, THE RONALD PRESS CO. N.Y. 1966; U.S. MAIL SCHEDULE

Exhibit 5—Computer microfilm vs paper continuous forms output: Substantial savings in storage and shipping costs

## Cost reductions for all users

In summary, then, by superimposing the costs of service centers (Exhibit 2) on those of in-house installations (Exhibit 3), it can be seen that the use of a computer microfilm recorder will always result in the lowest data processing cost at all levels of usage (Exhibit 6).

These facts should be apparent:

1. A computer microfilm service center is always about 50 percent cheaper than a paper service center, and this cost advantage probably will go higher.
2. A computer microfilm service center is the least expensive data processing alternative up to about 200,000 pages of output per month. (200,000 pages per month is the maximum output of a single shift working six days per week on a 360/30 with one attached line printer.)

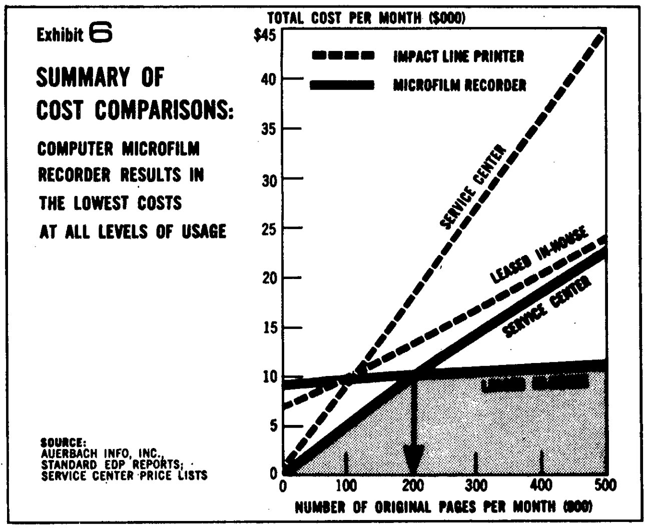


Exhibit 6—Summary of cost comparisons: Computer microfilm recorder results in the lowest costs at all levels of usage

3. Beyond 200,000 pages of output per month, an in-house computer with a microfilm recorder is by far the least expensive data processing alternative.
4. An in-house computer/microfilm recorder can bring about a cost saving vis-a-vis a com uter/line printer installation beyond about 90,000 to 100,000 pages per month, or only five to six hours of computer time per day, with paper output.

Thus, if decisions regarding an in-house capability versus utilization of a service bureau were always rational and financially sound, 100 percent conversion from paper to computer microfilm output could be expected. To anticipate a conversion ratio of 100 percent is, of course, unrealistic. Nonetheless, the pricing revolution which the computer microfilm service companies have brought about in the data processing service industry should result in very extensive use of the computer microfilm recorder in this segment of the computer industry. The small data processing user will be the primary beneficiary of the dramatic reduction in data processing service bureau costs. Similarly, medium-scale to heavy computer users will find the substantial cost and operating advantages of an in-house recorder sufficiently compelling to bring about heavy conversion to microfilm output in this market segment.

*Microfilm is the most efficient medium for storing and accessing generated computer data*

Computer microfilm is actually, by a wide margin, the most efficient and economical storage and retrieval system for computer generated information. Microfilm has always been superior to paper from a bulk handling and storage standpoint. With the introduction of the computer microfilm recorder, it can now also approximate electronic time-sharing systems in performance for the great majority of information storage and retrieval applications. Thus, computer output on microfilm can provide a simple, fast information system far superior to those currently in use. Indeed, computer microfilm service bureau managements indicate that it is not the substantial cost advantage of film over paper computer output which is most attractive to prospective customers, but rather its usefulness as an effective information system. The dramatic cost benefits, however, can be an extremely effective sales too, in getting the customer to consider microfilm seriously.

## Microfilm joins the computer era

Development of the computer microfilm recorder has brought in its wake a flurry of product development activity aimed at greatly facilitating access to information on microfilm. Most individuals think of microfilm only as an archival medium—for storing outdated information for which a need might or might not arise at some time in the future. Actually, the active use of microfilm for the storage and retrieval of information in daily use has been practiced by some pioneering users and companies for years. For the most part, these have been extremely large users (e.g., Social Security Administration). We feel that in large part the reluctance to adopt active microfilm systems has been due to the fact that information in such systems had to be manually sorted, updated, and coded—a tedious and time-consuming task.

Now, however, this task has been eliminated through the development of computer microfilm coding systems which can provide manual access to one page out of 73,500 in one to five seconds.

Additionally, the speed and ease with which computer information can be obtained on microfilm has been increased from days to literally minutes. One manufacturer has adopted a marketing program stressing "on time" information rather than "real time", which is, in fact, an accurate description. There is virtually no computerized information which cannot be obtained overnight in a fully useful, properly indexed format.

In sum, the user of computer microfilm has access to a "poor man's time-sharing" information system, as some have termed it, with no addition to his CPU costs.

**MICROFILM VIEWERS vs. TIME SHARING SYSTEMS:** Exhibit 7
**FAVORABLE COMPARISON IN TERMS OF COST AND SPEED**

| | TERMINAL COST | | TIME |
|---|---|---|---|
| **MICROFILM VIEWERS** | TERMINAL ONLY | TYPICAL RENTAL PER MONTH (Includes Modem) | TIME TO DISPLAY FULL PAGE (8,000 Characters) |
| HF IMAGE CARD SYSTEM | $2,000-$3,450 | $95-$160 | 1-4 seconds |
| Stromberg DatagraphiX 1700 (Automatic Magazine) | $ 1,248 | $ 42 | 4-15 seconds |
| KODAK PVM (Manual Roll) | $ 600 | $ 21 | 8-20 seconds |
| **COMPUTER TERMINALS** | | (Includes Maintenance) | |
| SANDERS 720 (Video) | $ 9,025 | $468 | 0.2 seconds |
| IBM 2265 VIDEO (Plus Controls) | $15,000 | $478 | 3.1 seconds |
| IBM SELECTRIC 2741 | $ 3,100 | $130 | 533 seconds (9 min.) |
| TELETYPE KSR 33 | $ 450 | $ 90 | 800 seconds (13 min.) |
| **COMPUTER COMMUNICATIONS** | APPROXIMATE TRANSMISSION COST | | TIME TO TRANSPORT FULL PAGE (8,000 Characters) |
| HIGH SPEED – BROAD BAND (TELPAK-D) | $ 45/mile/month | | 0.1 seconds |
| WATS SERVICE | $240 - $2,000/month | | 27 seconds |
| DEDICATED VOICE | $ 2 - 3/mile/month | | 27 seconds |
| LO SPEED | $ 1/mile/month | | 800 seconds (13 min.) |

[1]COMPUTER COMMUNICATIONS COSTS MUST BE INCLUDED WITH TERMINAL AND STORAGE COSTS IN TIME SHARING SYSTEMS.
SOURCE: AUERBACH INFO. INC., COMMUNICATIONS REPORT; AUERBACH INFO. INC., STD EDP REPORTS; COMPANY PRICE LISTS

Exhibit 7—Microfilm viewers vs time sharing systems: Favorable comparison in terms of cost and speed

## Computer microfilm competes effectively against time-sharing

Many feel that time-sharing will become the most common method of providing access to computer generated information. But, it can be shown that for most applications, the storage and retrieval of information electronically is very uneconomical relative to a computer microfilm system.

For example, in one specific application, a data storage capacity of 15,000 pages, to be updated daily, was required. The effective cost of this application on a commercial time-sharing disc file system equalled about $3.00 per page or a total of $45,000 per month. On microfilm, this same information can be updated once a day for approximately $0.60 per page or $9,000 per month—a storage cost reduction of almost 80 percent. Moreover, the time-sharing system would incur additional costs for terminal connect time and computer search time.

Therefore, we feel that microfilm, as a medium of access to computer information, will become much more commonly employed than time-sharing in the future. Time-sharing, however, will always be required for applications in which immediate interaction with the data base is desired.

Microfilm permits the storage cost savings just described because it has a significantly greater storage density capacity than the magnetic storage media used in time-sharing systems (i.e., disc packs and data cells). While it is only possible to store approximately 1,050 bits per square centimeter on computer magnetic materials, it is possible to store 1,000 times this amount; or over one million bits, on a square centimeter of microfilm.

In addition to storage costs, the relative disadvantages of time-sharing for information storage and retrieval include substantially higher terminal and communications costs (Exhibit 7).

As shown in the exhibit, a full page of information can be accessed in one to four seconds on the CARD device. To equal this speed with a time-sharing system a high-cost video terminal and Telpak-D communications line must also be employed.

As a result of these cost factors, microfilm is the more economical of the two systems for most normally encountered information storage and distribution problems. The surface illustrated in Exhibit 8 delineates the points (determined by file size, number of users, and update frequency) at which a microfilm system is roughly cost equivalent to an electronic information storage and retrieval system.

For the problems located within the surface, a microfilm system is less expensive; for those outside the surface, electronic systems are less expensive.

For example, the exhibit demonstrates that when information must be available to 200 users and updated every business day, a microfilm system is more economical for files of 14,000 pages or less. A file of this size could contain the daily closing stock quotations for the NYSE, ASE, and OTC market for over four years. Similarly, a 14,000 page file could contain all the records for payroll, personnel, and finished goods inventory (plus 10,000 accounts receivable records) for an average industrial corporation with sales of $800 million per year.[3]

There are two types of commonly encountered applications for which microfilm is not a suitable replacement for time-sharing: when the user wishes to input, manipulate, and extract data at will, and when updating is required more than once a day, such as in transportation **reservation** systems (these cases are

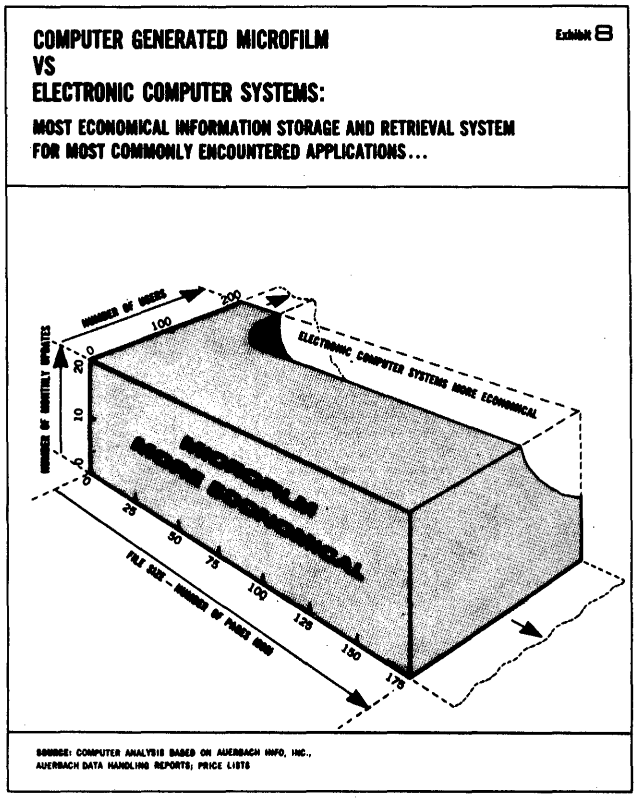


Exhibit 8—Computer generated microfilm vs electronic computer systems: Most economical information storage and retrieval system for most commonly encountered applications

located above the update frequency = 20 times/month plane in Exhibit 8). Whereas time-sharing allows information stored in a computer to be updated immediately and made readily available in updated form to all users, with a microfilm system four to six hours is the minimum time one may expect for file update, preparation, and distribution.

However, in most other commonly encountered information storage and retrieval applications, computer processed data is required for informational purposes only, such as in referencing records to service a customer inquiry. In these cases, a microfilm information system is equally as effective and far less expensive than a time-sharing system.

Recently, hybrid information systems have been introduced in which a data base is stored on microfilm while recent updates and changes can be retrieved electronically from computer memory. These systems, which utilize the advantages of both microfilm and time-sharing systems, should find widespread acceptance in the future.

## REFERENCES

1 *Report on non-impact printing project*
Army Materiel Command Jan 1968
2 *The computer and the small company*
Auerbach Info. Inc.
3 *Statistical Abstract of the U S 1968 (89th edition)*
U S Bureau of Census
Washington D C 1968 and
*The computer and the small company*
Auerback Info. Inc.

# Design of distributed communications system—A case study

*by* N. NISENOFF

*Computer Command and Control Company*
Washington, D. C.

## INTRODUCTION

The development of a concept for a Department of the Army Civilian Personnel Management and Manpower Data Reporting System and an Optimum Automatic Data Processing System was undertaken by Computer Command and Control Company in June, 1967.*

The work was initiated by the Department of the Army to meet the increasing demand for more detailed information about civilian employees, as required in connection with Army-wide civilian personnel career management programs, and in view of new and more detailed general governmental reporting requirements. In addition, the system was to be capable of maintaining data concerning the wide range of skills and experience of Army personnel. A further goal was the reduction to a minimum of the time delay in communicating relevant personnel data for the purpose of applicant screening.

The system, as developed, is a generalized civilian personnel information system that embraces all aspects of the Army's civilian personnel management activity and control. It provides the information gathering, processing, storing, querying and reporting capabilities to meet the requirements of Headquarters, Department of the Army; all echelons of field commands; the Department of Defense; the Bureau of the Budget; the Civil Service Commission; and other government agencies.

* This effort has been performed for the Deputy Chief of Staff for Personnel, United States Army, under contract DAHC15 67 C 0265.

The design concept:

1. Provides a powerful, efficient, open-ended, processing capability at a cost level that is the minimum commensurate with the system requirements.
2. Utilizes the most advanced (yet proven) hardware and information entry, storage and retrieval techniques available as so to effect data entry, validation, distribution, storage and organized retrieval with minimal human intervention.
3. Offers direct, rapid, complete and easy exchange of both formatted and unformatted personnel information among authorized individuals and offices at all levels.
4. Provides standardized funtional personnel management information formats and processing techniques, together with adequate on-line analytical tools.
5. Makes exchange of data with the Civil Service Commission, the Department of Defense and with other Army systems simple and easy, providing data definitions have been standardized.

Insofar as practicable, use has been made of present data bases. By applying automatic file conversion techniques previously developed, it will be possible to efficiently convert many existing data bases into random access files that can be electronically updated and queried. Particular attention has also been given to the problems of interfacing with and making the best use of existing automated or partly automated general management information systems within the Department of the Army.

The scope of the project is in part indicated by the following: There are over half a million civilian employees of the Army paid from appropriated funds, of whom about 140,000 are foreign nationals. In addition, there are about 200,000 civilian employees overseas who are paid from non-appropriated funds but who are administered by Civilian Personnel Offices. To service these employees, there are some 200 Civilian Personnel Offices scattered around the world. For just the United States Civil Service employees, it is estimated that approximately two billion characters of information will need to be carried in the Army Civilian Personnel Management and Manpower Data Reporting System.

Following the initial data gathering and analysis phase, the record and file structuring effort was undertaken. During this phase all candidate data elements were identified, classified and organized into files. File usages were then examined and files were assigned to appropriate storage media. For example, the data elements required to develop a reply to a relatively frequent query were placed in a fast mass random access storage subsystem. On the other hand, any data element required but infrequently was placed in the magnetic tape storage subsystem.

Given the results of the file structuring study, four separate and distinct Continental United States hardware configurations were postulated, and two additional prepared for overseas components. Each configuration is capable of performing the data processing functions required. The Continental United States configurations specified were:

1. A centralized single computer system;
2. A regionalized five computer system;
3. A decentralized twelve computer system; and
4. A localized twenty-one computer system.

A cost analysis was then performed to evaluate each configuration. The overall results of this evaluation, including the cost of initial loading of the data bank, is shown in Figure 1.

After considering this cost data plus the other advantages and disadvantages, Configuration I was selected. It of is interest to note that the total cost of the system per Civilian Personnel Office for Configuration I is about the same as the salary and overhead cost of one GS-5.**

To establish the practicality of the implementation of the proposed system, a break-even analysis was performed. A reduction in work force of four percent

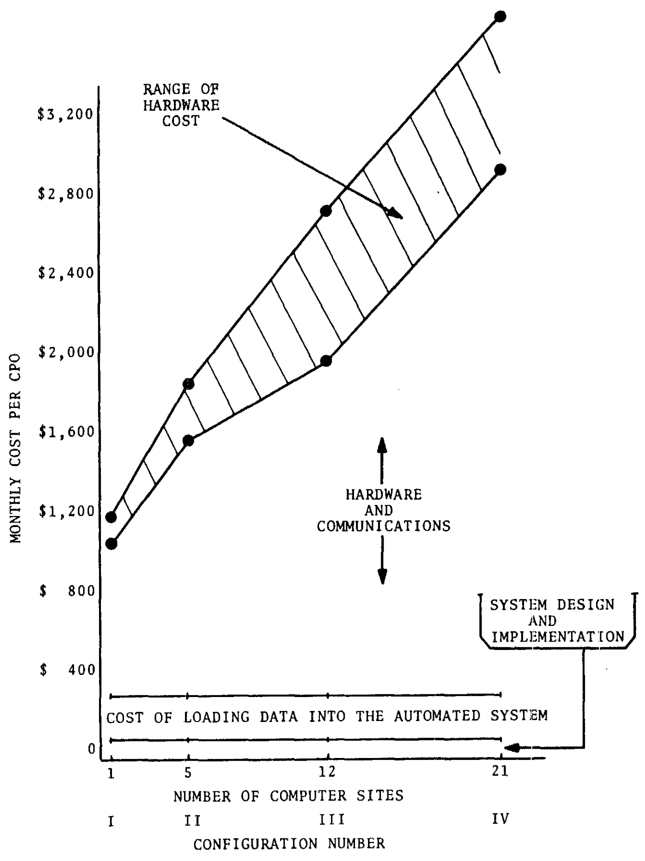


Figure 1—Monthly cost per CPO for four examined configurations

within the Civilian Personnel Offices is the break-even point, while a six percent reduction would produce net savings of approximately 1.25 million per year. There is evidence that in an automated system, this reduction in work force could be made with no loss of efficiency or productivity. In fact, the automated system could be expected to greatly increase staff efficiency and productivity, as well as provide management information vastly improved with respect to timeliness, completeness, accuracy and internal consistency. Finally, analytical services would be available which cannot be achieved with a manual system.

The software and programming aspects of the overall problem were not examined as thoroughly as desired. Certain assumptions were made, among these were:

1. The computer hardware would be dedicated to the application.
2. Computer manufacturer's software support would be adequate for all needs except specific applications packages.
3. The query language and the storage and retrieval subsystems were not specified.

** At the time the report was prepared, a GS-5 earned $5,732.00 per year.

Fortunately, the study was not performed without prior experience or knowledge concerning these points. Previous efforts by the Company, as well as members of the team, had been concerned with these very points. Estimates were made and employed.

A subsequent investigation*** required a more detailed and thorough examination of these very points. The results of that study will be reported upon in the near future.

*The dimensions of the problem*

### General description

Within the Department of the Army, military and civilian personnel administration is centered in single offices, at all levels, wherever both military and civilian personnel are found in significant numbers. However, at the execution level, branching is noted between the military and civilian personnel staffs in executing day-to-day detailed, direct operational responsibilities. This will continue to be the case in the future.

Table I indicates the distribution of civilian employees with respect to citizenship; by Army area in the Continental United States, or geographical area outside Continental United States, and membership of the staff of Army Material Command, Corps of Engineers or "other" organizations. Additionally, it presents the number of Civilian Personnel Offices servicing the ten designated groupings.

### Information and processing requirements

As a basic premise, it is assumed that processed num-

*** Contract No. FA68WA-1913, Design of FAA Manpower and Personnel Information System.

TABLE I—Distribution of civilian employees and civilian personnel offices

| *Army or Geog. Area* | *No. of Orgs.* | *No. of CPO's* | *Total U.S. Employees* | *Foreign Nationals* | *Grand Total* |
|---|---|---|---|---|---|
| I | 87 | 63 | 147,800 | | 147,800 |
| III | 34 | 22 | 57,200 | | 57,200 |
| IV | 25 | 20 | 45,200 | | 45,200 |
| V | 37 | 30 | 63,600 | | 63,600 |
| VI | 29 | 25 | 42,600 | | 42,600 |
| Hawaii | 7 | 1 | 5,900 | | 5,900 |
| Alaska | 6 | 3 | 3,100 | | 3,100 |
| Far East | 4 | 5 | 5,100 | 81,000 | 86,100 |
| Europe | 19 | 16 | 7,600 | 56,200 | 63,800 |
| SOCOM | 2 | 2 | 1,900 | 2,800 | 4,700 |
| Totals | 250 | 187 | 380,000 | 140,000 | 520,000 |

*Breakdown of U.S. Employees*

| *Army or Geog. Area* | *AMC* | *Corps of Engineers* | *Other* | *Total U.S. Employees* |
|---|---|---|---|---|
| I | 72,400 | 14,300 | 61,100 | 147,800 |
| III | 19,700 | 10,700 | 26,800 | 57,200 |
| IV | 16,400 | 7,000 | 21,800 | 45,200 |
| V | 31,700 | 8,500 | 23,400 | 63,600 |
| VI | 19,800 | 6,700 | 16,100 | 42,600 |
| Hawaii | | | 5,900 | 5,900 |
| Alaska | | 500 | 2,600 | 3,100 |
| Far East | | | 5,100 | 5,100 |
| Europe | | 300 | 7,300 | 7,600 |
| SOCOM | | | 1,900 | 1,900 |
| Totals | 160,000 | 48,000 | 172,000 | 380,000 |

eric, textual and graphic information delivered to the user must be adequate to meet both predicatable and ad hoc needs Processing and delivery must be timely.

The basic parts of such an automated system are:

1. A central processor (or processors).
2. Data storage capacity and retrieval capability.
3. A means for inputting and outputting information.
4. Adequate data communications.

### Central processor

In discussing central processors, there are two basic factors to consider. If there is to be but one central processor, then the only requirement of importance is that it have the capacity and time available so as to be able to handle the input and output loads and perform the required processing.

On the other hand, if a multiple computer configuration is decided on, in addition to the requirement set forth above, there has to be a distinct set of software programs for each local or decentralized computer type which is not internally compatible with the master computer, plus additional programs to provide for transfer of data from one computer to another. This adds considerably to both the cost and the complexity of the system.

If the central processor is not dedicated to civilian personnel use, but is shared, the particular priority that would most probably be accorded civilian personnel information processing would entail delays of indeterminate length. As the number of computers handling civilian personnel information is increased, the likelihood of sharing the computer increases greatly. At the same time, any procedure that involves the output of more than one computer would not be completed until time is available on the last available computer. It is not only the delay that can prove to be vexing; it is also the fact that it is most difficult to ascertain how long the delay might be.

### Data storage capacity and retrieval capability

When, as is expected, 40 percent of all employees are in the career management program, storage for approximately 1.8 billion characters of information will be required for the records of the United States Army civilian personnel. Storage for an additional 600 million characters will be necessary for United States employees overseas and foreign national employees.

### Data elements

In determining how large an individual record would be, it is recognized that the record of a new employee will not be as extensive as that of an employee who has worked many years. To measure that difference, the number of characters for a typical personnel record of a GS-1 through GS-5, of a GS-6 through GS-11 and a GS-12 through GS-18 were recorded.

Table II is a summary of information concerning the data elements which are required to meet both present and anticipated needs. It also provides the numbers of characters required for each of three record categories.

### Input and output requirements

There are 160 Civilian Personnel Offices in the Continental United States, which will require a total of from 300 to 400 input/output consoles, depending on the make or type finally chosen. The overseas Civilian Personnel Offices will require an additional 60-90 consoles.

Both from qualitative and cost standpoints, the effects of proper or improper console selections will clearly be very significant in view of the large number of units involved.

Four broad categories of information are present within the system.

1. Information necessary to update the files and records.
2. Queries and responses.
3. Statistical data.
4. Processing outputs in general.

Detailed analysis of data presently handled or required in updating files and records and in meeting all except ad hoc query demands results in 20 characters/man/day of data input and 18 characters/man/day output. To handle ad hoc queries and their response, an additional 12 characters/man/day (six input, six output) are considered adequate. These statistics are based on there being 240 working days a year. Table III shows the known inputs to and outputs from the central processor and the numbers of characters/man/year and characters man/day.

#### *Alternative configurations*

As a starting point, a single computer system was considered. Next, an integrated system of six computers was examined, then one with 13, and finally, one with

TABLE II—Summary information: Data elements

| Data Elements | Maximum Record Size (Char) | Average Sized Record: GS 1–5 Plus W. B. | | Average Sized Record: GS 6–11 Plus W. B. Supervisors | | Average Sized Record: GS 12–18 | |
|---|---|---|---|---|---|---|---|
| | | Single Computer | Multiple Computer | Single Computer | Multiple Computer | Single Computer | Multiple Computer |
| Organization Elements* (0) | | | | | | | |
| Position Elements (P) | 164 | 164 | 164 | 164 | 164 | 164 | 164 |
| Civilian Personnel Elements (CP) | 11,132 | 2,796 | 2,796 | 4,335 | 4,335 | 6,192 | 6,192 |
| Career Management Program Elements (CM) | 7,548 | | | 2,562# | 7,548# | 2,456# | 7,548# |
| Executive Assignment** Elements (EA) | 2,467 | | | | | | |
| Primary Personnel (PP) Elements | 139 | | 139 | | 139 | | 139 |
| Statistics (ST)* | | | | | | | |
| TOTAL | | 2,960 | 3,099 | 7,061 | 12,186 | 8,812 | 14,043 |

Using the statistical data tabulated below, the average record length would be 4,700 for a single computer system and 7,100 for a multicomputer installation.

| | |
|---|---|
| GS 1–5 plus W. B. less W. B. Supervisors | —230,000 employees = 60% |
| GS 6–11 plus W. B. Supervisors | —113,000 employees = 30% |
| GS 12–18 and PL Appointees | — 37,000 employees = 10% |

(U.S. citizens only)

* The storage requirements for organizational and statistical elements are insignificant as compared with the personnel data storage requirements. Therefore, no values are assigned.

** Executive assignment data not carried in Army System but by Civil Service Commission; presented here for information only.

# The character count for average record length in the CM category is kept at maximum record length since it is expected that personnel in this program will have been in the civil service for several years and therefore will have need for all the record space available.

22. The process halted at this point because it was becoming apparent that the complexities of a multicomputer system—and, as a result, the unpredictable time responses as well as costs—were growing at a far higher rate than any advantages that might accrue.

The first configuration involves a single computer located in the Washington area. Figure 2 shows a schematic of this. Each Civilian Personnel Office would input to and receive data from this one computer. A complete record concerning each civilian working for the Department of Army would be maintained in this computer, except for foreign nationals in Europe and the Far East. This would result in a total of approximately 380,000 records.

At each Civilian Personnel Office, there will be at least one input/output console. Here, all information would be entered that would update personnel records, as well as Civilian Personnel Office queries. Each console would also recieve all query-response outputs as well as general information outputs and record printouts that are needed for all personnel management purposes.

TABLE III—Major inputs to and outputs from system (Annual)

| *INPUTS* | *No. of Characters* |
|---|---|
| Payroll Change Slip (2515) | 150 |
| Request for Personnel Action (52) | 300 |
| Application for Federal Employment (57) | 750 |
| Request for Referral List (2302-2) | 350 |
| Installation Training (750) | 1,000 |
| Employee Performance & Career Appraisal (2302-4) | 1,600 |
| Employee Performance Rating (1052) | 100 |
| Job Description Rewrite (374) | 400 |
| Qualification Record (2302) | 100 |
| Total Inputs | |
| *OUTPUTS* | 4,750 |
| Referral List Response (2302-2) | 165 |
| Notification of Personnel Action (50) | 300 |
| Career Employment Record (2302-5) | 3,600 |
| Position Review (275) | 100 |
| Occupational Inventory of Civilian Positions (1629) | 100 |
| Table of Distribution & Allowances (2952) | 70 |
| Civilian Personnel by Basic Rate (3100) | 70 |
| Civilian Personnel Employment Report (3250) | 22 |
| Total Outputs | 4,427 |

| | | |
|---|---|---|
| Known Inputs | 4,750 = | 20 Characters/man/day |
| Known Outputs | 4,427 = | 18 Characters/man/day |
| Known Requirements | 9,177 = | 38 Characters/man/day |

For ad hoc queries and presently unanticipated requirements, 6 characters/day each for input and output are added.

For system concept development the following values are used:

| | |
|---|---|
| Total Inputs | 26 Char/man/day |
| Total Outputs | 24 Char/man/day |
| Total Requirements | 50 Char/man/day |

The system will be able to accept inputs of authorized manpower spaces and changes to them as these allocations or changes to specific spaces are made. By comparing the data with position information reported by Civilian Personnel Offices, it will be easy to determine, at any time, discrepancies between vacancies and established positions.

The second configuration provides for five computers carrying personnel records in addition to the RAPID system computer complex in Washington. Figure 3 is a schematic diagram of this.

These five processors would be located, with numbers of Army and Civilian Personnel Offices serviced, as shown in Table IV.

Personnel records of employees assigned to each Civilian Personnel Office would be stored in the computer for the respective Army area. Each Civilian Personnel Office would use its consoles to communicate with the computer in the Army area where it was located, in the same manner as described previously for the operation of a single computer system.

These five area computers would be connected electrically to the RAPID computer in Washington. The central (Washington) computer would store some 20 critical elements of information (approximately 100 characters) concerning each non-career employee plus additional career management information for each employee in the career management system.

This master computer would thus be able to produce statistical data as well as provide responses to many queries without requiring access to the area computers. Answers not obtainable from the RAPID system

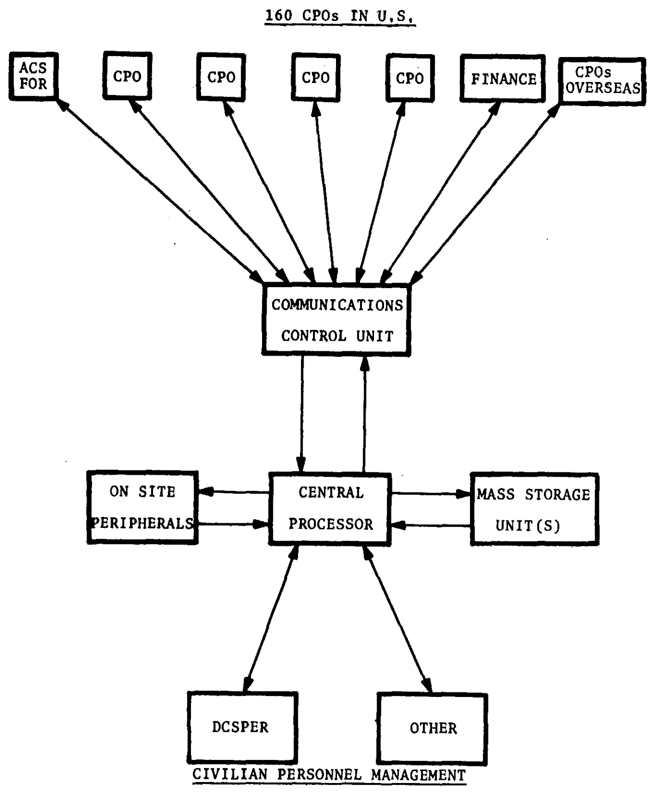


Figure 2—Single computer configuration data input and output

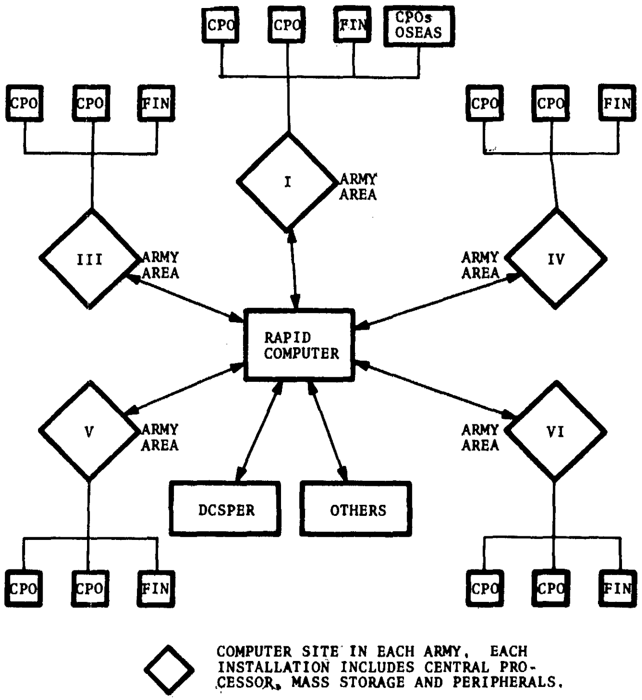

CPO
CPO
FIN
CPOs OSEAS
I
ARMY AREA
III
ARMY AREA
IV
ARMY AREA
RAPID COMPUTER
V
ARMY AREA
VI
ARMY AREA
DCSPER
OTHERS
FIN
COMPUTER SITE IN EACH ARMY, EACH INSTALLATION INCLUDES CENTRAL PROCESSOR, MASS STORAGE AND PERIPHERALS.

Figure 3—Six computer configuration

computer complex would be generated by polling the computer(s) able to furnish the answer, or if the specific computer containing the desired information was not known, all five computers would be queried.

The third configuration provides for twelve computers carrying (primarily) personnel records, plus the RAPID system computer in Washington. Figure 4 shows a schematic diagram of this.

In this approach, a configuration was developed wherein the two major employers of civilian personnel, the Army Materiel Command and the Corps of Engineers, retained the records of their own personnel in their own computers. The balance of the employees are serviced in Army area computers as in Configuration II.

In an effort to store the data as close to the Civilian Personnel Offices as feasible, a twenty-two computer configuration was also postulated and studied. A schematic of this is shown in Figure 5. Again, the RAPID system computer would serve the same

TABLE IV—Six computer configuration description

| *Command* | *Location* | *Personnel Serviced* | *Number of CPO's* |
|---|---|---|---|
| DCSPER | Pentagon | | |
| First Army* | Ft. Meade, Maryland | 163,000 | 86 |
| Third Army | Ft. McPherson, Georgia | 57,000 | 22 |
| Fourth Army | Ft. Sam Houston, Texas | 45,000 | 20 |
| Fifth Army | Ft. Sheridan, Illinois | 63,000 | 30 |
| Sixth Army** | The Presidio, San Francisco, California | 52,000 | 29 |
| | | 380,000 | 187 |

* To include records for U. S. personnel in Europe, the Far East and Southern Command.

** To include records for U. S. employees in Hawaii and Alaska.

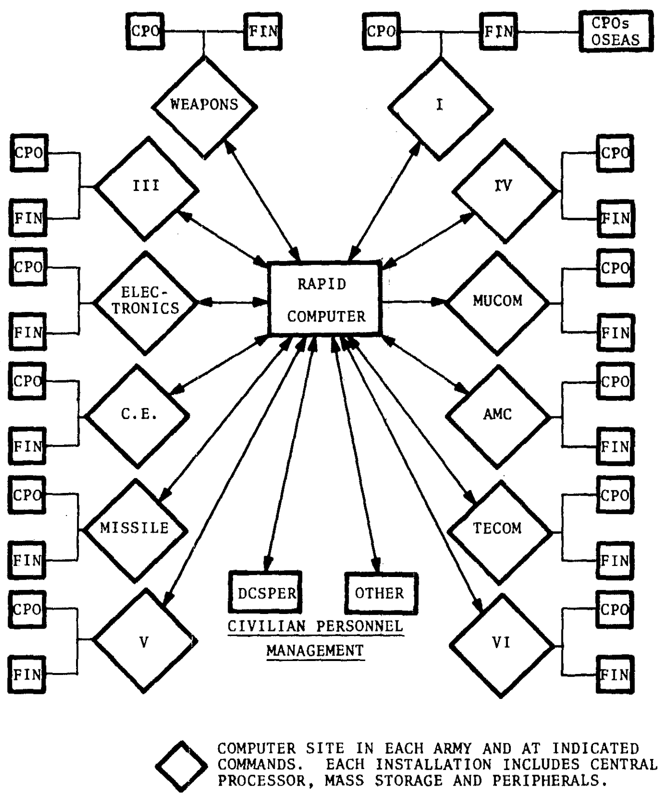

CPO
FIN
CPOs OSEAS
WEAPONS
I
III
IV
ELEC-TRONICS
RAPID COMPUTER
MUCOM
C.E.
AMC
MISSILE
TECOM
V
VI
DCSPER
OTHER
CIVILIAN PERSONNEL MANAGEMENT
COMPUTER SITE IN EACH ARMY AND AT INDICATED COMMANDS. EACH INSTALLATION INCLUDES CENTRAL PROCESSOR, MASS STORAGE AND PERIPHERALS.

Figure 4—13 computer configuration

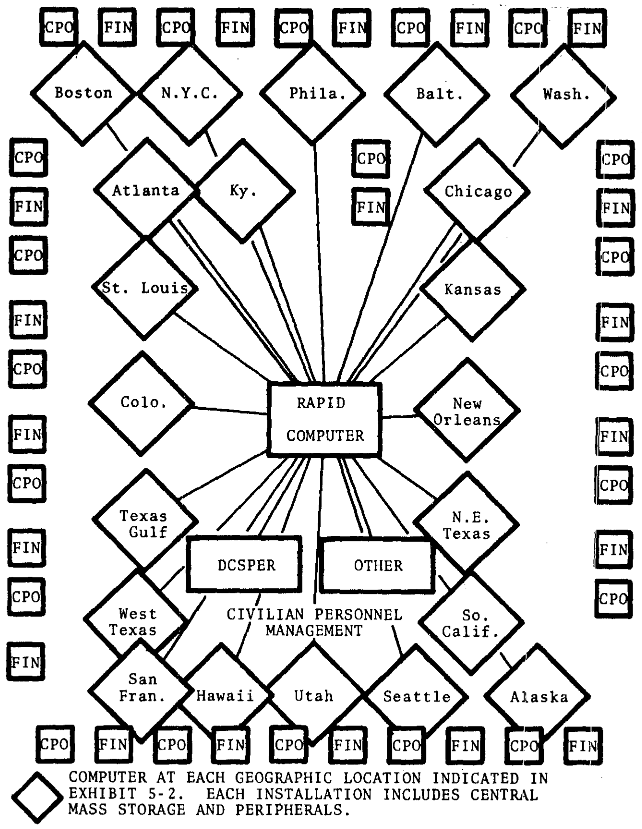


Figure 5—22 computer configuration

function as described in the explanation of Configuration II.

*Hardware technology*

The general considerations which must be taken into account in designing a complex system such as the one under discussion can be divided into two major areas. One is concerned with the hardware to be employed and the other with the software. The hardware area, in turn, can be subdivided into four parts, while there are two distinct aspects of software use to be considered.

The following discussion will be concerned only with the hardware aspects, specifically:

1. The central processor hardware considered during the study,
2. The mass random access storage systems,
3. The input/output terminals to be located at the various Civilian Personnel Offices and Headquarters offices throughout the Continental United States and,
4. The communications techniques or channels to be utilized.

Of the four specific hardware subsystems, the communication channels, the terminals and the computers are highly interrelated. Further, the communication software package either provided by the hardware manufacturer or developed by the contractor must be integrated in such a fashion as to permit the large volumes of data transfer to work in a smooth, well-integrated fashion. This was assumed to be true for this study.

## Automatic data processing systems

Following a detailed examination of the automatic data processing systems available at the time of the study (1967), a representative selection was made. Summary data concerning these systems will be found in Table V.

## Random access mass storage systems

Of the four configurations specified, Configuration I requires the largest and fastest mass random access storage subsystem. Under this configuration, an on-line storage capacity (at one computer site) of an estimated 1.8 billion bytes of information will be required. Today, no single device is available which can meet

TABLE V

| MANUFACTURER AND MODEL | DATE FIRST INSTALLED | CPU AND MEMORY | | | PRINTER | CARD READER | MAGNETIC TAPE | | | COMMUNICATIONS CONTROLLER | | | SOFTWARE | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | MEMORY RANGE AND CYCLE TIME | COST LEASE | COST PURCHASE | SPEED MODEL | SPEED MODEL | TRANS. RATE MODEL | COST LEASE | COST PURCHASE | MAXIMUM RATE MODEL | COST LEASE | COST PURCHASE | FULL | PARTIAL |
| RCA<br>SPECTRA 70/35 | 10/66 | 16-<br>65K BYTE<br>1.44μs | 4.2K | 210K | 1250 LPM<br>70/243-10 | 1435 CPM<br>70/237 | 60K C/S<br>70/442 | 900 | 42.3K | 2400<br>BAUD<br>70/668 | 900 | 45K | | X<br>NO MULTI-<br>PROC. |
| RCA<br>SPECTRA 70/45 | 7/66 | 16-<br>262K BYTE<br>1.44μs | 10.8K | 540K | 1250 LPM<br>70/243-10 | 1435 CPM<br>70/237 | 60K C/S<br>70/442 | 900 | 42.3K | 2400<br>BAUD<br>70/668 | 900 | 45K | | X<br>NO MULTI-<br>PROC. |
| RCA<br>SPECTRA 70/46 | 9/68 | 16-<br>131K BYTE<br>1.44μs | 14.6K | 811K | 1250 LPM<br>70/243-10 | 1435 CPM<br>70/237 | 60K C/S<br>70/442 | 900 | 42.3K | 6K B/S<br>70-668-31 | | | X | |
| BURROUGHS<br>B5500 | 11/64 | 4-<br>32K WORD<br>4.μs | 19K | 778.5K | 1040 LPM<br>B328 | 1400 CPM<br>B129 | 18-72K C/S<br>B425 | 850 | 38.2K | 2400 BAUD<br>B249 AND<br>B5480 | 855 | 44K | | X<br>NO<br>ASSEMBLER |
| BURROUGHS<br>B6500 | 1/68 | 16-<br>196K WORD<br>.6μs | 16.2K | 772K | 1040 LPM<br>9241 | 1400 CPM<br>9112 | 9-32K C/S<br>DUAL DR.<br>9381-2 | 900 | 43.2K | VERY HIGH<br>B6350 AND<br>B6350-1 | 1000 | 48K | | X<br>NO<br>ASSEMBLER |
| SDS<br>SIGMA 5 | 8/67 | 4-<br>131K WORD<br>.85μs | 14K | 542K | 1000 LPM<br>7445 | 900 CPM<br>7140 | 60K C/S<br>7321 | 280 | 10K | 1800<br>BAUD<br>7614 | 500 | 20K | | X<br>NO RPG<br>COBOL<br>6/68 |
| SDS<br>SIGMA 7 | 12/66 | 4-<br>131K WORD<br>.85μs | 35K | 775K | 1000 LPM<br>7445 | 900 CPM<br>7140 | 60K C/S<br>7321 | 280 | 10K | 1800<br>BAUD<br>7614 | 500 | 20K | X | |
| GENERAL ELECTRIC<br>435 | 9/65 | 16-<br>128K WORD<br>2.7μs | 10K | 417K | 1200 LPM<br>PRTZ01 | 900 CPM<br>CRZ201 | 15-42K<br>MTH301 | 590 | 26K | 2400 BAUD<br>DATANET<br>30 | 1500 | 95K | X | |
| IBM<br>360/40 | 5/65 | 16-<br>262K BYTE<br>2.5μs | 10.2K | 500K | 600-1100<br>LPM<br>1403 | 800 CPM<br>1402 | 30-180K C/S<br>2401 | 785 | 38K | 2702 | 850 | 40.8K | | X<br>NO MULTI-<br>PROC. |
| IBM<br>360/50 | 9/65 | 65-<br>262 K BYTE<br>2.μs | 13.7K | 684K | 600-1100<br>LPM<br>1403 | 800 CPM<br>1402 | 30-180K C/S<br>2401 | 785 | 38K | 2702 | 850 | 40.8K | | X<br>NO MULTI-<br>PROC. |
| IBM<br>360/65 | 3/66 | 131-1048K<br>BYTE<br>.75μs | 13.5K | 580K | 600-1000<br>LPM<br>1403 | 800 CPM<br>1402 | 30-180K C/S<br>2401 | 785 | 38K | 2702 | 850 | 40.8K | | X<br>NO MULTI-<br>PROC. |
| CONTROL DATA<br>3100 | 2/65 | 8-<br>32K WORD<br>1.75μs | 9.6K | 299K | 800-1000<br>LPM<br>501 | 1200 CPM<br>405 | 15-60K C/S<br>604 | 630 | 27.5K | 40.8K B/S<br>3316 | | | | X<br>NO MULTI-<br>PROC. |
| CONTROL DATA<br>3300 | 12/65 | 8-<br>32K WORD<br>1.25μs | 12.5K | 470K | 800-1000<br>LPM<br>501 | 1200 CPM<br>405 | 15-60K C/S<br>604 | 630 | 27.5K | 40.8K B/S<br>3316 | | | | X<br>MULTI-<br>PROC. 6/69 |
| CONTROL DATA<br>3500 | 3/67 | 8-<br>262K WORD<br>.8μs | 17K | 650K | 800-1000<br>LPM<br>501 | 1200 CPM<br>405 | 15-60K C/S<br>604 | 630 | 27.5K | 40.8K B/S<br>3316 | | | | X<br>AS OF<br>12/67 |

this requirement. To attain this storage volume a number of units must be integrated into the total system. For example, use of the RCA RACE unit, actually the cheapest available device on a dollar per byte basis, would still require three units to accommodate the total volume of data. To purchase these units (including their individual control units), will cost nearly half a million dollars, while rental would be about $12,000 a month.

Table VI presents a summary description of the most likely candidates for the mass random access storage systems. Note that this listing of devices includes the largest mass storage units that are presently available as well as smaller units which have been considered for utilization with the smaller decentralized centers described in Configuration IV. Figure 6 shows the storage capacity as the independent variable with the cost/performance ratio shown in cents per bytes stored.

## Communications channels

Initially, the U. S. Postal System was considered as a valid technique of transmitting the daily accumulated data (from each Civilian Personnel Office) to the computer site. Further examination raised two objections. These were:

1. Cost involved.
2. Transmission delays.

The use of the Postal Service entails several costs which can be summarized, on a monthly basis, as:

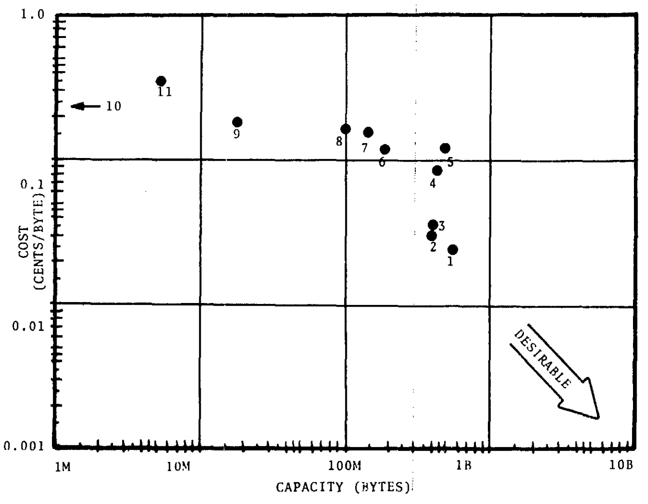
| | |
|---|---|
| Postage | $37,884 |
| Packaging | 233 |
| Addressing and Handling | 375 |
| Replacing Damaged and Lost Reels | 375 |
| | $38,867 or $240/ month/CPO |

TABLE VI

| MANUFACTURER NAME AND MODEL TYPE | CAPACITY | | STORAGE | | ACCESS | | ACCESS TIME: MILLISECONDS | | | PRICE* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | NUMBER OF BYTES | UNITS PER CONTROL | MEDIA | TRACKS PER SURFACE | NUMBER OF HEADS PER MECHANISM | TRACKS PER HEAD | MINIMUM | AVERAGE | MAXIMUM | PURCHASE | RENTAL | COST PER BYTE (¢/B) |
| RCA 568-11 (RACE) | 560M | 8 | STRIP | 128 | 16 | 16 | 20 | 200 | 385 | $145K | | .026 |
| IBM 2321 (DATA CELL) | 400M | 8 | STRIP | 100 | 20 | 20 | 25 | ~300 | ~600 | 136.5K | $2,800 | .034 |
| DATA PRODUCTS 5085 | 400M | 1 | DISK | 510 | 1 | 8 | 50 | 85 | 250 | 143K | | .036 |
| BRYANT 2AC 4000 | 419M | ? | DISK | 728 | 2 | 6 | 30 | 110 | 180 | 375K | | .089 |
| BURROUGHS 9375 | 500M | 1 | DISK | 150 | 150 | 1 | 0 | 60 | 120 | 590K | 9,900 | .118 |
| IBM 2314 | 210M | 1 | DISK | 200 | 1 | 18 | 25 | 75 | 135 | 250K | | .118 |
| CONTROL DATA 814 | 151M | 8 | DISK | 192 | 1 | 4 | 20 | 60 | 110 | 230K | 5,500 | .152 |
| UNIVAC FASTRAND 2 | 100M | 8 | DRUM | 12,480 | 1 | 64 | 39 | 92 | 154 | 165K | 3,750. | .165 |
| NCR 353-3 (CRAM) | 18M | 16 | STRIP | 56 | 56 | 1 | 24 | 235 | 235 | 35.5K | | .195 |
| SDS 7202 | 737K | 8 | DISK | 128 | 128 | 1 | 0 | 17 | 35 | 18K | | .245 |
| IBM 2311 | 7.3M | 1 | DISK | 200 | 20 | 10 | 25 | 75 | 135 | 26.3K | | .360 |

ARRANGED IN ORDER OF COST PER BYTE OF STORAGE

*Price for 1 (one) storage module only.



STORAGE UNIT LEGEND

| | | |
|---|---|---|
| 1. RCA 568-11 (RACE) | 5. BURROUGHS 9375 | 9. NCR 353-2 (CRAM) |
| 2. IBM 2321 (DATA CELL) | 6. IBM 2314 | 10. SDS 7202 |
| 3. DATA PRODUCTS 5085 | 7. CDC 814 | 11. IBM 2311 (DISC PACK) |
| 4. BRYANT 2AC 4000 | 8. UNIVAC FASTRAND II | |

Figure 6—Cost (cents per byte)/capacity (bytes)

For electrical communications, two distinct and different classification schemes or methods can be employed to facilitate the analysis. These are:

1. Governmental/non-governmental facilities.
2. TWX voice grade/broad band facilities.

Comparisons among costs for each of the services noted become very involved and complicated for a single configuration, let alone for four. However, each service is described below, following a brief discussion of the data volumes expected. With approximately 380,000 United States citizens covered by the system, and with a flow of forty to fifty characters per man per day over the communications channels; it seems almost mandatory that a "dedicated" communications system be available.

In the event personnel records are procesesd by a computer used for other applications as well, it is assumed that the personnel system will be available for major update processing and for query response during the third shift (eight hours). With an average of fifty characters/man/day traffic for 380,000 records, there will be a traffic flow of nearly twenty million bytes (eight-bit characters) a day. With a 70 percent line utilization, this requires a 1,000 bytes/second transmission capability.

## Autodin

Autodin can be used to provide the type of service required. However, the following points serve to eliminate it from consideration:

1. The service is not available at approximately 20 percent of the Civilian Personnel Office sites.
2. It is an extremely costly means of transmitting civilian personnel data. The charge is a fixed rate per site and is high, in part at least, because the system must be able to pass classified information. Since civilian personnel data would not be classified, except possibly for occasional specific information that would be afforded special handling, this costly apsect would not be needed.
3. Civilian personnel information would be afforded a low priority as compared to other data using Autodin. This would cause delays of variable and indeterminate length.

The cost, per site or terminal, depends upon the line bandwidth required, not the distance the message is sent or the line usage (time). These costs are shown in Table VII.

## Autovon

Autovon is a military leased, voice grade, direct dial telephone system. There is no apparent reason for not employing this system to transmit digital data during off-hours (6 p.m. to 6 a.m.).

Charges are variable, but an estimate of \$315-\$372/month/CPO seems reasonable.

Hard decisions concerning the use of Autovon for digital data transmission were not obtained, though statements were made that Autovon is used in some cases for data transmission. Neither could any indication be obtained that for night use lower or preferential rates were available.

TABLE VII—*Costs*

| *Autodin Bandwidth (Band)* | *Cost/Month Terminal Sit* |
|---|---|
| 75 | {1,188 |
| 150 | 2,375 |
| 1,200 | 9,504 |
| 2,400 | 14,250 |

## Wide area telephone service (WATS)

The most attractive data transmission channel studied during this effort, from the viewpoint of the Army Civilian Personnel Program, is the Wide Area Telephone Service (WATS). WATS offers two billing plans; a 24-hour, unlimited service, and a measured time service.

Under a measured time WATS contract, a basic monthly charge for the first ten hours of usage is made and an additional charge per hour of actual usage is levied. The tariff which governs this service is extremely detailed and a full discussion is beyond the scope of this report. However, a single computation will indicate the method of selecting between the unlimited and the measured WATS.

A single WATS line with a six band capability (full 48 state coverage), based in Washington, D.C. costs \$2,250 per month. The measured WATS, with the same capability, costs:

$$C = 370 + 29(H)$$

where,

C = Cost in dollars per month

H = Hours of usage beyond the first ten hours, per month.

The break-even point can be calculated by setting C = 2,250 and solving for H. This yields a value of H = 65 hours or 75 hours/month of circuit time or approximately three hours/day.

Definitive evaluation for the Army Civilian Personnel System must await final implementation decisions. However, a computation concerning the use of WATS for several possible configurations has been carried out, and is detailed in Table VIII.

## Leased broad band lines

Finally, use of leased broad band lines was examined. Their use was considered only for Configuration I implementation.

Many possible line linkages can be conceived. The one demonstrated here is for illustrative purposes only, but is typical.

Postulate that a concentration device, or subsystem (such as a very small digital computer with magnetic

TABLE VIII—Number of WATS lines required vs configurations

| *Configuration* | *Location* | *Number of Bands* | *Bands* | *Cost/ Month** | |
|---|---|---|---|---|---|
| I | Washington, D. C. | 4 | 1, 3, 6, 6 | $ 6,100 | |
| | | | | | $ 6,100 |
| II | Ft. Meade, Maryland | 2 | 1, 3 | $ 2,075 | |
| | Ft. McPherson, Georgia | 1 | 1 | 1,300 | |
| | Ft. Sam Houston, Texas | 1 | 1 | 2,015 | |
| | Ft. Sheridan, Illinois | 1 | 5 | 2,075 | |
| | Presidio, California | 1 | 2 | 2,000 | |
| | | | | $ 9,465 | |
| | PADIR System Input | 1 | 6 | $ 2,250 | |
| | | | | | $11,715 |
| III | Ft. Meade, Maryland | 1 | 3 | $ 1,575 | |
| | Ft. McPherson, Georgia | 1 | 1 | 1,300 | |
| | Ft. Sam Houston, Texas | 1 | 1 | 2,015 | |
| | Ft. Sheridan, Illinois | 1 | 5 | 2,075 | |
| | Presidio, California | 1 | 2 | 2,000 | |
| | Corps of Engineers, D.C. | 1 | 6 | 2,250 | |
| | AMC, D. C. | 1 | 6 | 2,250 | |
| | MUCOM, Edgewood, Maryland | 1 | 6** | 2,250 | |
| | T & E Com., Aberdeen, Maryland | 1 | 6** | – | |
| | WeapCom., Rock Island, Illinois | 1 | 5 | 1,500 | |
| | ECom., Ft. Monmouth, New Jersey | – | – | – | |
| | Missile Com., Huntsville, Alabama | – | – | – | |
| | | | | $17,215 | |
| | RAPID System Input | 1 | 6 | 2,250 | |
| | | | | | $19,465 |

* Includes intrastate charges, as required.
** Shared.

tape or disk), will be located at each Army Headquarters. A WATS system, similar to the system described for Configuration II (see Table VIII) will be installed. Leased broad band lines would aslo be installed from the five Army Headquarters to the RAPID site.

A broad band channel capable of transmitting 5,100 characters/second costs $15/mile/month. In addition each terminal requires a termination which rents for $250 each. As a result, monthly rental for the communication channels (and their terminations), but not the concentration, would be $18,465.

## Terminals

Terminals must have certain attributes. The requirements will vary, depending on whether the terminal is located (for example, at a major command headquarters, or in an operating Civilian Personnel Office) on the final network configuration selected, and on the communication means employed. However, certain minimal capabilities can be specified:

1. The data terminal must possess an "extended" (ASCII) keyboard, for entry of data.

TABLE VIII—(Contd)—Number of WATS lines required vs configurations

| *Configuration* | *Location* | *Number of Bands* | *Bands* | *Cost/ Month** |
|---|---|---|---|---|
| IV | Boston | 1 | X | $ 330 |
| | New York City | 2 | 1, X | 1,150 |
| | Philadelphia | 2 | 1, X | 875 |
| | Baltimore | 1 | X | 375 |
| | D. C. Area | 1 | 1 | 500 |
| | Atlanta | 2 | 1, X | 1,300 |
| | Kentucky | 2 | 1, X | 1,200 |
| | Chicago | 2 | 2, X | 1,475 |
| | St. Louis | 1 | X | 645 |
| | Kansas City | 1 | X | 610 |
| | Colorado | 2 | 2, X | 1,825 |
| | New Orleans | 2 | 1, X | 1,400 |
| | Texas Gulf | 1 | X | 815 |
| | NE Texas | 2 | 1, X | 1,815 |
| | West Texas | 2 | 1, X | 1,915 |
| | South California | 2 | 1, X | 1,850 |
| | San Francisco | 1 | X | 650 |
| | Hawaii | – | – | No information |
| | Utah | 1 | X | 500 |
| | Seattle | 2 | 1, X | 1,775 |
| | Alaska | – | – | No information |
| | | | | $21,005 |

X = Intrastate charges.

2. The terminal must have an extended storage ability to retain up to one day's input for transmission to the data processing site, with due allowances for peaks.
3. The terminal must be capable of receiving and storing information transmitted by the servicing computer (in general, during the night) in response to ad hoc requests or standing requirements.
4. The terminal must be capable of producing in hard-copy form all information transmitted from the computer site to the Civilian Personnel Office or other location either on-line, or on a delayed basis.
5. The unit must be able to communicate, via low cost (voice grade) telephone lines with the computer center.

After careful consideration of the characteristics of the many available principal Input/Output equipments, we have narrowed the field for further consideration to three. These are:

1. Mohawk 1103.
2. Dartex 1022.
3. Communitype 100SR.

### *The selection*

In performing the cost analysis for each of the four configurations examined, four different manufacturers' equipments were examined. These manufacturers' equipments were examined in the context of the processing loads required for both the centralized and distributed configurations. Table IX shows the specific central processors considered.

Although there are many approaches that can be followed to select a central processor for each of these configurations, one dominant constraint controlled the selections. This constraint was the requirement that the computer be able to utilize the amount of random access storage needed by the system at each location. Thus, although there are smaller computers available, not only from the four manufacturers whose equipments were examined, but also from other sources, the equipment selected represented, in general, the smallest computers that could do the job.

TABLE IX—Central processors employed in the cost analysis for each configuration

| *Manufacturer* | I | II | III | IV |
|---|---|---|---|---|
| CDC | 3304 | 3304 | 3114 | 3114 |
| BURROUGHS | 3501 | 3501 | 2501 | 2501 |
| IBM | 350/40 | 360/40 | 360/30 | 360/30 |
| RCA | 70/45 | 70/45 | 70/35 | 70/35 |

## Configuration I

Configuration I has a single centralized site into which all Civilian Personnel Offices address their data. Detailed examination showed that approximately 1.8 billion bytes of information would be the largest amount of dedicated random access storage required at this location. Supporting that subsystem would be a high-speed disk subsystem of approximately 100 million bytes. This subsystem would act as a directory and contain a high-use skeletal record for each employee whose total record was contained in the random access storage system. The central processor at this site is provided with approximately 65,000 bytes of high-speed core storage, a normal complement of standard peripherals; such as, a high-speed printer, a card reader and punch, a communications control unit as well as eight high-performance magnetic tape units. The magnetic tape units are employed to maintain an on-line journal of all system actions. They also act as replicate security repositories of current information in the event of equipment destruction, electrical information losses, over-writing, etc. In addition, the magnetic tape drives can be employed for other purposes during non-civilian personnel operations at the site. Finally, magnetic tape would be used to store trailer information (overflow beyond single fixed-format record storage capcaity) and archival information.

Communications with the centrally located computer site can be handled by four WATS lines. Two of these would cover the 48 states, one for the Eastern one-fourth of the United States, and one for the Eastern Seaboard. The extent of these lines, that is, the number of WATS bands, have been selected to provide the optimal coverage of CONUS based upon the geographic distribution of Civilian Personnel Offices and the populations they support.

Finally, each Civilian Personnel Office was examined to determine the number of terminals required. A keyboard, hardcopy printer, and an intermediate storage capability are considered a mandatory requirement for this system application.

To better understand the operation of the Configuration I system, consider the requirements for data transmission from and into a Civilian Personnel Office on a daily basis; 26 characters per man per day (on the average) are inputted to the computer from a Civilian Personnel Office, while 24 characters per man per day are outputted from the computer to a Civilian Personnel Office. With these figures, an estimate of the communication requirements can be made. Similarly, the estimate of the actual keyboard typing or data outputting can be obtained.

Taking all these facts plus the data provided in an earlier section into account, it can be shown that the cost for Configuration I will be in the order of $130,000 to $150,000 rental per month.

## Configurations II, III and IV

Configuration II is schematically represented in Figure 3. Configuration II represents a total of five computer sites and thus the amount of rental required to support these sites does increase. Similarly, the communication cost rises from approximately 6.1 to 11.7K dollars per month.

Configurations III and IV have been treated in the same fashion as Configuration II.

## Detailed Configurational Comparison

The results developed thus far may now be applied to the crucial problem—which of the four postulated configurations is recommended and why. The central processors selected for examination with respect to the four configurations postulated have been specified in Table IX.

In presenting a detailed description and price comparison, Table X summarizes the key requirements, i.e., the number of personnel serviced, mass random access storage capacities, and the data transmission volumes for each of the four configurations. These data were employed as guides in the hardware selections. Detailed equipment specifications and pricing/rental were also examined and are summarized in Table XI.

TABLE X—Summary of requirements for the four configurations examined

| Configuration Number | I | II | III | IV |
|---|---|---|---|---|
| Number of Computer Sites | 1 | 5 | 12 | 11 |
| People Serviced | | | | |
| Maxima (Excluded from Range Figures) | 380K | 163K | 48K, 75K, 83K | 47K, 58K |
| Minimum | | 45K | 11K | 2K |
| Range (Not including Maxima) | | 45–63 (4) | 11K–27K (9) | 2K–31K (19) |
| Average for Range | | 54K | 19K | 14K |
| RAM Capacity Requirements (Bytes) | | | | |
| Maxima (Excluded from Range Figures) | 1.8G | 655M | 225M, 350M, 390M | 223M, 180M |
| Minimum | | 162M | 52 | 15M |
| Range (Not including Max'ma) | | 162M–251M | 52M–127M | 15M–118M |
| Average for Range | | 201M | 90M | 62M |
| RAPID Supplemental RAM Capacity Required | 0 | 1.2G | 1.2G | 1.2G |
| Data Transmission Volumes (Characters per Second) | | | | |
| Maxima (Excluded from Range Figures) | | 8.2M | 2.4M, 3.6M, 4.2M | 2.4M, 2.9M |
| Minimum | | 2.3M | 0.6M | 0.1M |
| Range | 19M | 2.3M–3.2M | 0.6M–1.4M | 0.1M–1.6M |
| Average for Range | | 2.7M | 1.0M | 0.7M |

Two key aspects of the information contained in Table XI have been plotted to provide a clearer view. These are:

1. Hardware comparisons (exclusive of communications and terminal costs) for the four manufacturers, for each of the four configurations.
2. Comparison of monthly rentals for all hardware aspects (using an average set of values for the on-site computers and their conventional peripherals).

Point 1 is amply described in Figure 7, while Point 2 is presented in Figure 8.

At this point, Configurations III and IV were dropped from further consideration. The few advantages which could be enumerated in their favor were not sufficient to outweigh the added costs.

The selection between Configurations I and II appears less clear cut. Although the monthly rental for Configuration II is approximately 70 percent greater than the monthly rental for Configuration I, other factors must be examined. Only then can a decision be made.

In favor of Configuration I implementation are:

1. Lower monthly rental.
2. File centralization in one physical location close to Department of the Army and Department of Defense headquarters activities.
3. No undesirable redundancy in either hardware, software or machine processing. Also, if this processor is identical with that of the present RAPID system, then each can act as back-up for the other.
4. Availability of a "dedicated" computer for Army Civilian Personnel record-keeping. This implies that a self-established priority system can be employed.
5. A minimum of highly skilled ADP programmers, operator personnel, etc., required.
6. Data base "timeliness" and uniformity.
7. No limitations on "cross servicing."

TABLE XI—Purchase and rental comparison—four computer manufacturers and four system configurations

| Configuration | I | | II | | III | | IV | |
|---|---|---|---|---|---|---|---|---|
| Subsystem | Purchase M $ | Rental K $ * | Purchase M $ | Rental K $ * | Purchase M $ | Rental K $ * | Purchase M $ | Rental K $ * |
| Computer Site-Mass RAM | | | | | | | | |
| CDC | 1.0M | 26K | 4.6M | 115K | 9.6M | 226K | 14.4M | 384K |
| Burroughs | 0.8 | 16 | 2.6 | 60 | 5.6 | 132 | 10.1 | 231 |
| IBM | 1.1 | 17 | 4.0 | 83 | 8.7 | 179 | 15.4 | 316 |
| RCA | 1.2 | 26 | 4.7 | 96 | 8.6 | 196 | 17.0 | 345 |
| Mass RAM (Including RAPID Supplement) | | | | | | | | |
| CDC | 0.9M | 19K | 1.7M | 41K | 2.4M | 43K | 4.1M | 70 |
| Burroughs | 2.4 | 40 | 2.1 | 67 | 3.4 | 55 | 7.5 | 107 |
| IBM | 0.8 | 17 | 1.9 | 40 | 3.5 | 65 | 4.4 | 95 |
| RCA | 0.6 | 12 | 2.0 | 40 | 3.6 | 65 | 4.4 | 95 |
| Communications Channels-WATS | — | 6K | — | 12K | — | 20K | — | ~ 21K |
| Terminals | 3.5M | 88K | 3.5M | 88K | 3.5M | 88K | 3.5M | 88K |
| Totals | | | | | | | | |
| CDC | 5.5M | 138K | 9.7M | 255K | 15.6M | 371K | 22.0M | 563K |
| Burroughs | 6.6 | 149 | 10.5 | 227 | 13.5 | 294 | 21.2 | 447 |
| IBM | 5.3 | 131 | 8.9 | 221 | 15.7 | 351 | 23.3 | 519 |
| RCA | 5.3 | 131 | 10.1 | 235 | 16.7 | 378 | 24.8 | 549 |

*—Monthly
M—Millions
K—Thousands

On the other hand, Configuration II provides:

1. Local, autonomous control at the Army level of each computer system.
2. Redundancy of equipment which offers an alternative processing site in the event a system is down.
3. With lower processing loads per machine, cost sharing could be practiced.

No numerical weighting of these advantages seems appropriate. However, after a thoughtful review of each point, and a careful summation of all the points concerning each alternative, one is left with but one reasonable choice—Configuration I.

## SUMMARY

The investigation demonstrated that a highly distributed, Automated Personnel and Manpower System was feasible and would be cost-effective. It also showed, rather forcefully, that although the terminals were located throughout the country, a single concentrated

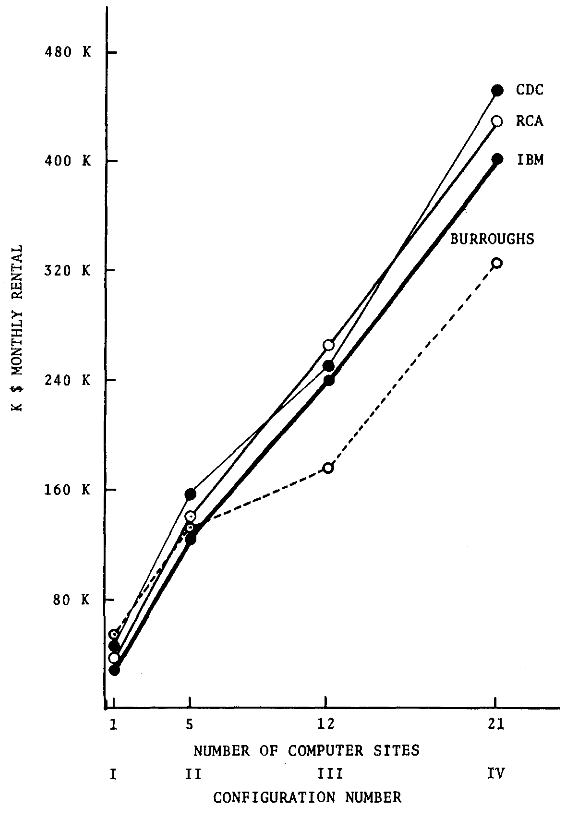


Figure 7—Monthly rental of CONUS computer systems for four manufacturers for all four configurations (Computer hardware only)

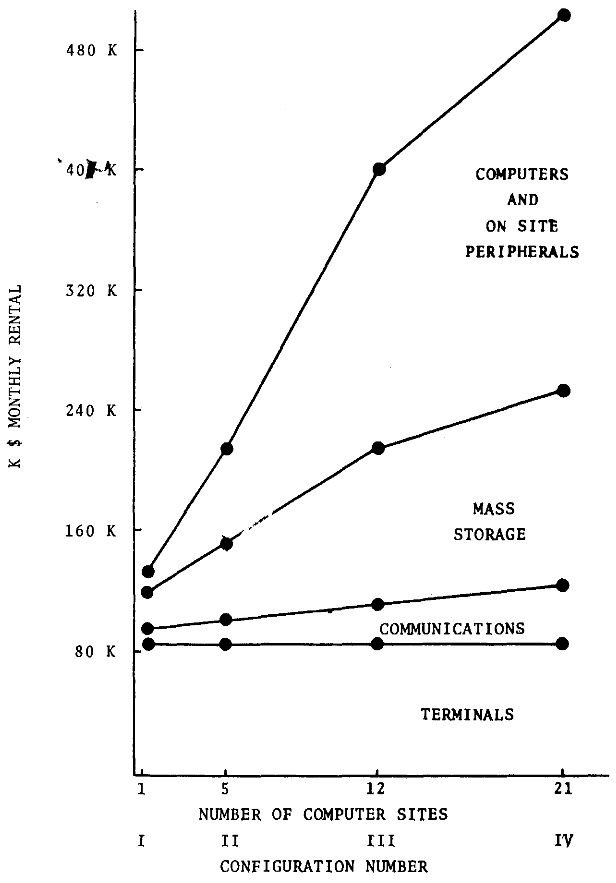


Figure 8—A typical total monthly rental breakdown for all four configurations (CONUS Only)

central processing site was by far the most economical approach to the system implementation.

An interesting fallout of the study was the fact that the cost of the communications channels required to support the system accounted for only one and one-half to three percent of the cost of the fully-implemented system.

Finally, the broadest result of the study was the conclusion that real time, on-line (or quasi on-line) systems were practical, cost-effective and currently attainable.

## ACKNOWLEDGMENTS

The material presented in this paper has been almost completely drawn from a report* submitted to the Department of the Army in February, 1968. The report was prepared by Mr. H. H. Lowell, Mr. J. V. Heimark, Mr. G. A. Koehler, Mr. R. J. Gibbons and the author.

Particular appreciation is due to Miss L. Richard for all her help and assistance in preparing this paper. Without her help it would not have been submitted.

* Final Report, "Civilian Personnel Management and Manpower Information System Concept for Department of the Vmy," 28 February 1968, H. H. Lowell, J. V. Heimark, G. A. Koehler, N. Nisenoff, R. J. Gibbons, Computer Command and Conrrtol Company.

# Analysis of the communications aspects of an inquiry-response system

*by* J. S. SYKES

*Bell Telephone Laboratories, Incorporated*
Holmdel, New Jersey

## INTRODUCTION

In order to meet the information retrieval needs of various industries, inquiry-response systems are being implemented by storing large data bases in centralized computer files. In some systems, the files are accessed by personnel primarily as the result of telephone calls from customers. As an example, in the airlines industry, computer files are accessed by reservation clerks to determine the availability of reservations for a specific flight. In this example, and in similar applications involving queries or requests from customers, input messages requesting certain information are generated by a customer representative and then transmitted to a computer from an input-output terminal such as a visual display device. When the computer has obtained the requested information, a response message is transmitted back to the requesting terminal, and the representative continues her dialogue with the customer.

For an inquiry-response system to function properly, the system must be designed to meet two grade-of-service, or performance, objectives. One objective is concerned with the interval a customer must wait before his call is answered by a representative. The other objective is concerned with the interval a customer must wait during the conversation until the customer representative can secure the necessary in formation from the computer; the naturalness of the dialogue degenerates as the retrieval time* increases.

* In this analysis, the retrieval time is defined as the interval from the time an input message is generated until the complete response has been received.

In order to meet the first objective, sufficient representatives must be available to handle the incoming voice traffic. To meet the second objective, an adequate data communications subsystem and sufficient computer processing capability must be provided.

In this paper an analytical model is presented that approximates the interaction of the voice and data communications subsystems in an inquiry-response system. The model can be used during preliminary investigations to gain insight and to obtain conservative estimates of communications capabilities required in order for a system to meet specified grade-of-service objectives. The model consists of relationships that involve basic communications parameters such as the following:

a. Rate at which calls are received from customers
b. Interval required for representatives to handle incoming calls
c. Number of input and corresponding computer response messages generated as the result of a customer call
d. Data transmission rates to and from the computer
e. Lengths of input and response messages.

The model uses these relationships in order to estimate the following quantities:

1. The number of customer representatives required in order to handle a given volume of offered calls at a specified grade of service
2. The volume of data traffic generated as a result

of the incoming voice traffic

3. The number of equivalent active terminals that can be served by a data link while meeting a specified retrieval time objective; an estimate of the retrieval time as a function of carried data traffic is used to obtain this quantity.
4. The number of data links required in order to meet a specified retrieval time objective
5. The average occupancy of the one or more data links serving the input-output terminals.

For illustrative purposes, the analytical model is applied to a hypothetical inquiry-response system. Both half-duplex** and full-duplex** methods of operation are considered for the data communications subsystem. For this example, estimates of average retrieval time are obtained with mathematical queuing models.

### *Assumed system characteristics*

The analysis and its application presented in this paper are based on assumptions concerning the incoming voice traffic, the characteristics of the data communications subsystem connecting the input-output terminals to the computer, and the computer processing capability. These assumptions are considered in this section.

The basic assumptions that have been made concerning the origination and nature of the voice traffic are the following:

1. The overall system is in a state of statistical equilibrium.
2. Calls are generated by customers in accordance with a Poisson distribution, which implies a large group of potential customers.
3. Durations required for representatives to handle incoming calls*** are distributed according to a negative exponential probability law.
4. Calls are answered immediately when there is a customer representative not currently engaged in a conversation; all other calls experience delay.
5. Delayed calls are answered in a first-come, first-served order as representatives become free.

** With half-duplex operation, message transmission is allowed in either direction, but not both directions simultaneously; simultaneous transmission in both directions is called full-duplex operation.

*** These durations would consist of the talking time with the customer plus subsequent time (if any) required to perform call-related tasks.

The assumed overall configuration of the voice-access network as well as the data communications subsystem is illustrated in Figure 1. The voice-access network is assumed to consist of the established telephone network that provides line-switched connections from the customer to the business location. Calls are automatically routed to an idle representative unless they must be delayed; if so, the call distributor maintains the calls in a queue until a representative becomes free.

The data communications subsystem is assumed to consist of a group of input-output terminals such as visual display devices that are associated with a common control unit, which is connected to a computer by means of a data link. Various methods of operation are possible for this data communications configuration. These possibilities depend on whether or not message transmission is allowed simultaneously in both directions, whether or not the computer requests traffic by means of polling, whether the polling characters are directed to individual terminals or to a control unit that gathers input messages from all of the terminals, etc. This analysis considers both full-duplex and half-duplex methods of operations. Polling of the control unit by the computer and multimessage transmissions in each direction are assumed.

Computer processing time, as used in this paper, refers to the overall interval from the instant an input message* enters the multiprocessing computer until the corresponding response is placed in queue for transmission back to the requesting customer representative. Thus, processing time includes input message analysis, data retrieval from one or more memory files (perhaps even from another computer), and response preparation; in addition, the processing times may be prolonged by queuing delays within the computer. It has been assumed that an estimate of the average computer processing time for a system is available; as will be explained, this estimate is used in determining the average retrieval time.

### *System analysis*

In this section the analytical model of the communications aspects of an inquiry-response system is developed.

* Examples of input messages are initial inquiries, requests for page flips, and any subsequent inquiries generated during a customer's call. In addition, in some systems, update messages may be sent to the computer, perhaps after a call has been terminated. If so, it is assumed that for each updating message the computer returns some type of acknowledgment.

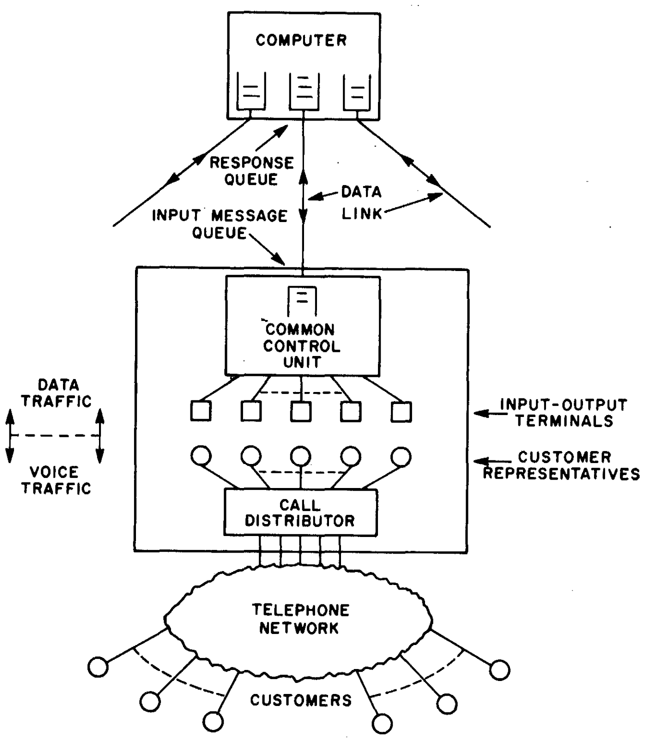


Figure 1—Inquiry-response system

## Personnel required to handle offered voice load

Assume that during the period of maximum incoming customer calls, i.e., the system busy hour, the calls are received at a rate $\lambda_v$. Assume further that the average duration required for representatives to handle incoming calls is $\overline{V}$. The voice load $\overline{E}_v$ handled by the representatives is therefore given by

$$\overline{E}_v = \lambda_v \overline{V}, \tag{1}$$

where $\overline{E}_v$ is commonly expressed in erlangs, a dimensionless unit. The number S of personnel required to handle $\overline{E}_v$ erlangs during the busy hour is dependent on the grade of service G(x) to be offered customers, i.e., the promptness with which customers' calls would be answered. An example of G(x) is the following: at least 0.95 of the customers' calls should be answered within x = 20 seconds from the time ringing begins.

If the assumptions previously stated concerning the voice traffic are met, values of S can be obtained for a specified G(x) by using the following formulas:[1]

$$\begin{aligned} G(x) &= 1 - \text{Prob[Answering Delay} > x \text{ secs]} \\ &= 1 - P[D > x] \\ &= 1 - \left\{ P[D > 0] \exp \frac{-x(S - \overline{E}_v)}{\overline{V}} \right\}; \end{aligned}$$

$$0 \leq \overline{E}_v < S \tag{2}$$

where P[D > 0], commonly called the Erlang C function, is given by

$$P[D > 0] = \frac{\dfrac{(\overline{E}_v)^S}{(S-1)!\,(S-\overline{E}_v)}}{\displaystyle\sum_{n=0}^{S-1} \frac{(\overline{E}_v)^n}{n!} + \frac{(\overline{E}_v)^S}{(S-1)!\,(S-\overline{E}_v)}},$$

$$0 \leq \overline{E}_v < S \,. \tag{3}$$

Equation (2) is a result of A. K. Erlang's exponential holding time analysis. A summary of his analysis along with various delay curves was published by E. C. Molina.[2] For specified values of S, values of P[D > 0] are tabulated in Reference 1 as a function of the ratio $\overline{E}_v/S$.

## Conversion of offered voice load to data traffic

The amount of data traffic generated as the result of a customer call is a random variable. Some calls may involve only one or possibly two input messages and the associated responses. Other calls, which may be multipurpose, may require six or eight such interactions; in addition, some updating of the computer files may be involved. In this paper, $\overline{I}$ will be used to represent the average number of interactions generated as the result of a call.

Let $\lambda_i$ represent the average rate during the busy hour at which input messages are generated by the group of customer representatives served by one data link. By using $\overline{I}$, $\lambda_i$ can be related to $\lambda_v$ as follows:

$$\lambda_i = \overline{I}\lambda_v \,. \tag{4}$$

Let $\lambda_r$ represent the average rate during the busy hour at which corresponding response messages are prepared by the computer and placed in the output queue for the data link. Since it is being assumed that each input message to the computer results in a response, the average rates $\lambda_i$ and $\lambda_r$ are equal.

The second factor influencing the volume of generated data traffic is the average time $\overline{t}_i$ required to transmit a message to the computer. This quantity is the quotient of

$\bar{\ell}_i$ = the average number of characters that comprise messages transmitted to the computer, and

$\tau_i$ = the rate of transmission from the control unit to the computer, i.e.,

$$\bar{t}_i = \frac{\bar{\ell}_i}{\tau_i}. \quad (5)$$

Correspondingly, $\bar{t}_r$, the average time required to transmit response messages from the computer to the control unit is given by

$$\bar{t}_r = \frac{\bar{\ell}_r}{\tau_r}. \quad (6)$$

The product of $\lambda_i$ and $\bar{t}_i$, which will be denoted by $\rho_i(\tau_i)$, represents the erlangs of data traffic generated during the busy hour for transmission at a rate $\tau_i$ from the control unit to the computer. Likewise, the product of $\lambda_i$ and $\bar{t}_r$, which will be denoted by $\rho_r(\tau_r)$, represents the erlangs of response data traffic transmitted at a rate $\tau_r$ from the computer to the control unit during the busy hour.

With full-duplex message transmission, separate one-way transmission facilities carry $\rho_i(\tau_i)$ and $\rho_r(\tau_r)$. Therefore, expressions for the magnitudes of $\rho_i(\tau_i)$ and $\rho_r(\tau_r)$ (in erlangs) can be independently determined by using Equations (1), (4), (5), and (6), i.e.,

$$\rho_i(\tau_i) = \lambda_i\bar{t}_i$$

which leads to

$$\rho_i(\tau_i) = \frac{\bar{E}_v\bar{I}\bar{\ell}_i}{\bar{V}\tau_i}. \quad (7a)$$

Similarly, since it is being assumed that $\lambda_i = \lambda_r$,

$$\rho_r(\tau_r) = \frac{\bar{E}_v\bar{I}\bar{\ell}_r}{\bar{V}\tau_r}. \quad (7b)$$

With half-duplex message transmission, the same facility is alternately used for input and output traffic. Therefore, $\rho_i(\tau_i)$ and $\rho_r(\tau_r)$ can be combined to give $\rho_{tot}(\tau_i, \tau_r)$, i.e.,

$$\rho_{tot}(\tau_i, \tau_r) = \rho_i(\tau_i) + \rho_r(\tau_r)$$
$$= \frac{\bar{E}_v\bar{I}}{\bar{V}}\left[\frac{\bar{\ell}_i}{\tau_i} + \frac{\bar{\ell}_r}{\tau_r}\right]. \quad (8)$$

Let $\alpha$ represent the ratio of the average length of response messages to the average length of input messages; if $\tau_i = \tau_r$, Equation (8) then reduces to

$$\rho_{tot}(\tau_i, \tau_r) = \frac{\bar{E}_v\bar{I}\bar{\ell}_i}{\bar{V}\tau_i}[1 + \alpha]. \quad (9)$$

In summary, Equations (7), (8) and (9) reveal the manner in which the various communications parameters affect the amount of generated data traffic.

## Volume of data traffic allowed per data link

As indicated by the notation, calculated erlang values obtained for $\rho_i(\tau_i)$, $\rho_r(\tau_r)$, and $\rho_{tot}(\tau_i, \tau_r)$ are based on specified transmission rates. Erlang values are not sufficient by themselves, however, to determine the number of data kinks operating at the assumed rates that would be required to implement the data communications subsystem. For example, if $\rho_{tot}(\tau_i, \tau_r)$ were less than one erlang, it could be inferred that one data link would suffice for that traffic. However, in order to avoid excessive storage usage and extended retrieval times, data links cannot be used to their full capacity. In fact, as will be shown, the average retrieval time increases without bound as the average occupancy of a data link approaches unity; average data link occupancy refers to the average portion of the busy hour that the data link is being used for message transmission.

Although data link occupancy must be limited, it is desirable to use data links as efficiently as possible. Let $\hat{\rho}_{tot}(\tau_i, \tau_r)$ denote the maximum volume (in erlangs) of data traffic that can be carried by the data link operating at transmission rates $\tau_i$ and $\tau_r$. For half-duplex operation, $\hat{\rho}_{tot}(\tau_i, \tau_r)$ is numerically equal to $\rho_{\max}(\tau_i, \tau_r)$, where $\rho_{\max}(\tau_i, \tau_r)$ represents the maximum allowable occupancy for a data link operating at rates $\tau_i$ and $\tau_r$.

For full-duplex operation, $\hat{\rho}_{tot}(\tau_i, \tau_r)$ is the sum of $\hat{\rho}_i(\tau_i)$ and $\hat{\rho}_r(\tau_r)$, which represent the maximum data volume (in erlangs) that can be carried on the input and output links operating at rates $\tau_i$ and $\tau_r$, respectively. For systems in which the input and output traffic volumes are unequal, the average occupancy of the input and output links may be considerably different. For this case,

$$\rho_{\max}(\tau_i, \tau_r) = \mathrm{Max}[\rho_{\max}(\tau_i), \rho_{\max}(\tau_r)],$$

where $\rho_{\max}(\tau_i)$ and $\rho_{\max}(\tau_r)$ correspond to $\hat{\rho}_i(\tau_i)$ and $\hat{\rho}_r(\tau_r)$, respectively. For a full-duplex system, $\hat{\rho}_{tot}(\tau_i, \tau_r)$ is not a constant but is dependent on the ratio $\bar{t}_r/\bar{t}_i$.

The value of $\hat{\rho}_{tot}(\tau_i, \tau_r)$ for a particular subsystem is governed by the specified retrieval time objective for the system. A commonly used objective is as follows: The average retrieval time should be $\overline{T}_{\max}$ seconds or less. Another type of objective* can be expressed similar to the voice traffic grade of service G(x), e.g., 0.95 of the retrievals should be received within T′ seconds.

Either a computer simulation or analytical means can be used to determine values of $\hat{\rho}_{tot}$ $(\tau_i, \tau_r)$ that correspond to a specified retrieval time objective. With a properly written simulation, one can obtain probability distributions as well as all moments of interest. However, using a simulation can be costly during preliminary investigations in which one is studying the effects of various communications parameters on the retrieval time. For this reason, a well-formulated mathematical queuing model can be useful and rewarding for these investigations, even though results from queuing models that represent complex systems are often limited to average values.

## Number of input-output terminals allowed per data link

An important consideration in the communications design of an inquiry-response system is the maximum number of active input-output terminals, or equivalently the maximum number of active personnel, that can be served by a particular data link without exceeding a specified retrieval time objective. This maximum, which will be denoted by $S_{\max}$, is obviously related to $\hat{\rho}_{tot}(\tau_i, \tau_r)$. In this section, a method is outlined for approximating values of $S_{\max}$ for specified values of the grade-of-service objectives and the other communications parameters. As will become apparent, the method may be used iteratively to determine which combinations of parameter values permit specified voice traffic and retrieval time objectives to be met. If costs are associated with these combinations, insight can be gained concerning which means of implementation is most economical.

The first step towards getting values of $S_{\max}$ is to use the specified values to construct graphs (using Equations (3) and (8), respectively) that show S versus $\overline{E}_v$ and $\rho_{tot}$ $(\tau_i, \tau_r)$ versus $\overline{E}_v$, where $\rho_{tot}(\tau_i,\tau_r)$ represents the sum of $\rho_i$ $(\tau_i)$ and $\rho_r(\tau_r)$ for both full-duplex and half-duplex cases. Corresponding points from these two graphs are then plotted to give a third graph showing S versus $\rho_{tot}$ $(\tau_i, \tau_r)$; the value of S corresponding to the point $\rho_{tot}(\tau_i, \tau_r) = \hat{\rho}_{tot}$ $(\tau_i, \tau_r)$ is $S_{\max}$.

The second step is to relate the values of $S_{\max}$ and the values of the retrieval time objective, e.g., $\overline{T}_{\max}$, that correspond to equal values of $\rho_{tot}$ $(\tau_i, \tau_r)$. Thus, a graph such as $S_{\max}$ verus $\overline{T}_{\max}$ can be constructed for given transmission rates $\tau_i$ and $\tau_r$. The benefit of such graphs can be increased considerably if the ordinate also shows the values of $\overline{E}_v$ that correspond to the values of $S_{\max}$. By using estimates of the expected voice load incoming to a cluster of customer representatives, the number of data links required to accommodate the cluster can be readily deduced from the graph for each specified retrieval time objective. This technique will be discussed further in the model application section.

Plots of $S_{\max}$ and $\overline{E}_v$ versus the retrieval time objective can aid investigations of the cost of implementing a system to meet a specified average retrieval time objective. For example, a designer may discover that for a relatively small increase in the allowable $\overline{T}_{\max}$, considerable savings in transmission and computer port costs could be achieved by serving more representatives with a single data link.

## Number of data links required

As was mentioned above, graphs showing $S_{\max}$ and $\overline{E}_v$ versus the retrieval time objective can be used to estimate $L(\tau_i, \tau_r)$, the number of data links required to interconnect the computer and the input-output terminals serving the customer representatives. There is also a more analytical method for estimating $L(\tau_i, \tau_r)$ in which values of $\hat{\rho}_{tot}(\tau_i, \tau_r)$ are used. The same general method can be applied to full-duplex and half-duplex message transmission subsystems; however, it should be remembered that for the full-duplex case, the value of $\hat{\rho}_{tot}(\tau_i, \tau_r)$ may change if the value of the ratio $\bar{t}_r/\bar{t}_i$ is changed.

Let k represent the ratio of the total volume of

* System studies are often desirable in the final design stages of a system to determine whether a given design will allow a specified percentile-type objective to be met. Because of mathematical complexity, however, analytical methods can seldom if ever be used for such studies; a simulation is normally required. For preliminary investigations, analyses based on average values can be used to obtain valuable insight concerning the sensitivity of the retrieval time to various system parameters. This insight can be very helpful in designing and running a subsequent simulation. Lack of such insight often results in very costly system simulations.

generated input and output data traffic (in erlangs) to $\hat{\rho}_{tot}(\tau_i, \tau_r)$, i.e., let

$$k = \frac{\rho_{tot}(\tau_i, \tau_r)}{\hat{\rho}_{tot}(\tau_i, \tau_r)}$$

Let K represent the integer part of the ratio k. The number of data links required to serve the cluster is given by

$$L(\tau_i,\tau_r) = 1 + K \tag{10a}$$

If K = k, i.e., k is an integer, then

$$L(\tau_i,\tau_r) = K$$

For half-duplex links, if it can be assumed that the total volume $\rho_{tot}(\tau_i, \tau_r)$ is divided evenly among them, the average occupancy of each link is given by

$$\rho = \frac{\rho_{tot}(\tau_i. \tau_r}{L(\tau_i, \tau_r)} \tag{10b}$$

*Model application*

In this section, the analytical model is applied to a hypothetical information retrieval design problem. For this example, it is assumed that information required for the operation of a business, such as customer service and billing records, is to be stored in a computer. Input-output terminals will permit access to the computer files; it is assumed that retrievals are primarily required in order to intelligently handle telephone calls from customers. Several clusters of input-output terminals are to be served by the same computer complex. The cluster to be considered in this example is concerned only with information retrieval; it is assumed that file modifications are done by other personnel.

The basic configuration proposed for this cluster is illustrated in Figure 1. Telephone calls from customers are routed by the automatic call distributor to idle customer representatives. Each customer representative is equipped with an input-output terminal. These terminals are associated with a common control unit, which is connected to the computer by means of a data link.

One objective of this analysis is to determine the basic requirements of the data communications subsystem, i.e., how many common control units in conjunction with their data links are required to accommodate the number of customer representatives that will be needed to handle the incoming telephone calls? To help answer this question, both full-duplex and half-duplex methods of operation are considered. Following the description of these assumed methods of operation, representative parameter values are used to indicate how these two proposals can be quantitatively compared.

## Description of assumed methods of operation

The first assumed method of operation to be described involves half-duplex message transmission, which may have some economic advantages over full-duplex operation for some geographical configurations. Half-duplex operation is more suited for clusters generating and receiving relatively low data traffic volumes and for which retrieval time objectives are not critical. One disadvantage of half-duplex operation is the line time required to reverse the direction of transmission; this interval will be referred to as the reversal time.

The disadvantage of reversal times can be partially overcome if the computer polls and delivers groups of messages to the common control unit instead of single messages to the individual input-output terminals. This method of operation will be referred to as group poll and delivery as opposed to single poll and delivery operation. When a large number of terminals are served by a control unit, group polling significantly reduces the line time required for reversing the direction of transmission and transmitting polling characters. In addition, as the volume of data traffic increases, group polling lessens the variance of the interval from the time an input message is ready for transmission until it has actually been transmitted to the computer.

When the reversal time durations are comparable to message transmission times, data link efficiency is increased considerably by allowing multimessage transmissions for both input and response messages, i.e., priority is not assigned to either type of message. Line efficiency increases because a reversal is not required following the transmission of each lower priority message in order to check the status of the higher priority message queue.

With group polling and multimessage transmissions, all input messages generated by the terminals since the last poll are sequentially transmitted to the computer. Only when the input message queue becomes empty is the direction of transmission reversed. After the reversal, the computer begins delivering the queue

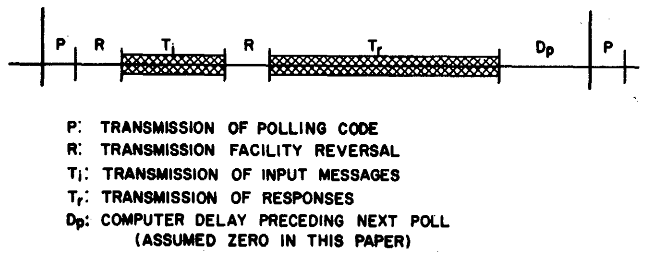

P
R
$T_i$
R
$T_r$
$D_p$
P
P: TRANSMISSION OF POLLING CODE
R: TRANSMISSION FACILITY REVERSAL
$T_i$: TRANSMISSION OF INPUT MESSAGES
$T_r$: TRANSMISSION OF RESPONSES
$D_p$: COMPUTER DELAY PRECEDING NEXT POLL (ASSUMED ZERO IN THIS PAPER)

Figure 2A—Typical cycle of operation (Half-duplex message transmission)

of responses to the control unit, which distributes each response to the appropriate terminal. After all responses have been delivered, the computer polls the control unit either immediately, or optionally after some specified delay* $D_p$, and the cycle repeats. A fixed number of characters that identify the control unit is sent preceding message transmissions from the control unit. This cycle of operation is illustrated in Figure 2A.

During a given cycle, either queue or even both queues can be found empty. If, for example, both are found empty, a group poll and delivery cycle degenerates to a polling sequence followed by a succession of reversal times, which are separated by a "No Traffic" character sequence that identifies the control unit. Such degenerate cycles are assumed to reoccur until at least one message accumulates in either the input or the response message queue.

With the full-duplex case, reversal times are unnecessary, since the control unit can be transmitting and receiving simultaneously. However, group polling of the control unit is still beneficial, since polling interference on the delivery line occurs less frequently. With full-duplex operation, input messages generated by the customer representatives are ordered in a first-come, first-served manner for transmission from the control unit. Transmissions to the computer begin immediately after a polling code is received from the computer; the polling codes are interspersed among messages received from the computer. All messages that have accumulated awaiting the polling code, as well as those that are generated during the transmission, are transmitted to the computer. An interval of duration** $D_p$ starts at the end of a transmission from the control unit; at the end of this interval the computer sends another polling code.

* For the half-duplex case, $D_p$ is assumed to be zero in this paper for the half-duplex case.

** For full-duplex operation, $D_p$ was assumed to be one second in this paper.

As soon as a response is prepared by the computer, it is entered into an output queue for delivery. It is transmitted immediately unless another transmission is already in progress; if so, the response is delayed until all responses ahead of it in the queue have been sent. Thus, while input messages are being transmitted by the control unit, response messages corresponding to previous input messages are being received by the control unit. Full-duplex message transmission is illustrated in Figure 2B.

## Personnel to handle incoming voice traffic

It will be assumed that the assumptions stated previously concerning voice-access subsystems apply to this example. It will further be assumed for this example that during the busy hour of the busy day the average number of calls per hour are not expected to exceed 600; the average duration of each call is expected to be approximately three minutes. By using Equation (1), it is found that $\overline{E}_v$, the expected voice traffic load, should not exceed 30 erlangs. In order to determine S, the number of customer representatives required to handle this traffic volume, a grade-of-service objective must be specified. In Figures 3A and 3B, S has been plotted as a function of $\overline{E}_v$. Figure 3A shows G(10), G(20), and G(30), where each is assumed equal to 0.95. Figure 3B shows the effect of varying the value of G(20) from 0.90 to 0.975.

Figures 3A and 3B reveal that the grade-of-service standard for answering voice calls can be improved considerably with the addition of a relatively few representatives. For example, assuming that the aver-

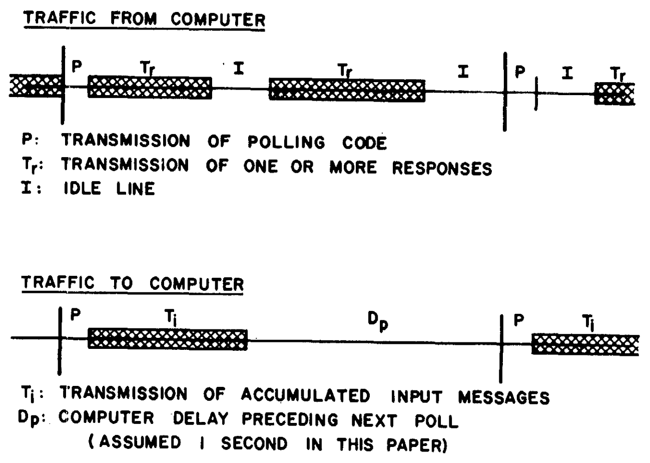

TRAFFIC FROM COMPUTER
P
$T_r$
I
$T_r$
I
P
I
$T_r$
P: TRANSMISSION OF POLLING CODE
$T_r$: TRANSMISSION OF ONE OR MORE RESPONSES
I: IDLE LINE
TRAFFIC TO COMPUTER
P
$T_i$
$D_p$
P
$T_i$
$T_i$: TRANSMISSION OF ACCUMULATED INPUT MESSAGES
$D_p$: COMPUTER DELAY PRECEDING NEXT POLL (ASSUMED 1 SECOND IN THIS PAPER)

Figure 2B—Typical cycle of operation (Full-duplex message Transmission)

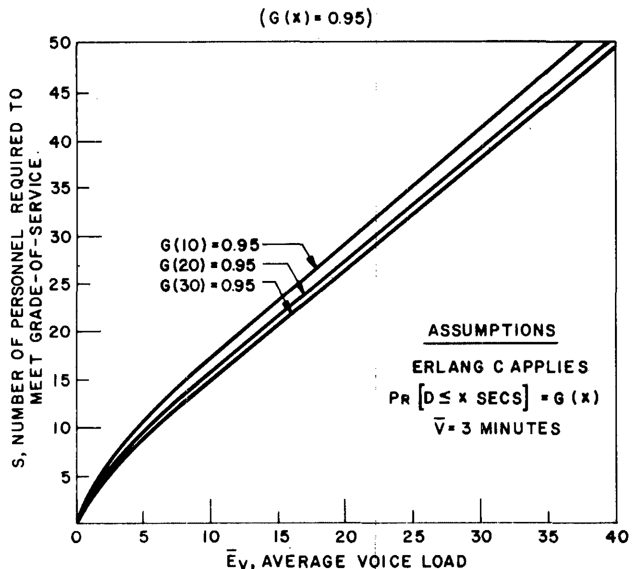


Figure 3A—Effect of grade of service on number of personnel required [G(x) = 0.95]

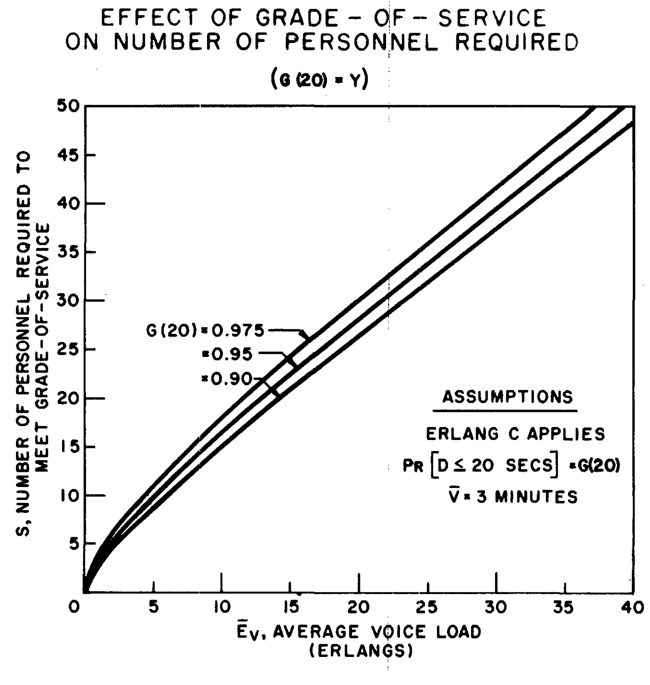


Figure 3B—Effect of grade of service on number of personnel required [G(20) = y]

age voice load during the busy hour is 30 erlangs and the grade-of-service objective is such that calls should be answered within 20 seconds, Figure 3B indicates that the fraction of calls that meet the objective can be increased from 0.9 to 0.975 by increasing the number of representatives from 38 to 42. These additional representatives could be individuals that are assigned as representatives only during busy hour conditions.

## Conversion of offered voice load to data traffic

The amount of data traffic generated is dependent on the degree of interaction between customer representatives and the computer. As an example, procedures could be outlined that would minimize the number of computer interactions per call by simply transmitting in a single response as much as possible of the information in a computer file. On the other hand, if the intent were to minimize the information that must be read by representatives, several interactions could be used during which the computer eliminated most of the undesired information. Computer processing limitations would favor the former method of operation; human factors considerations may favor the latter.[3]

An illustration of the effect of interactions on the amount of data traffic generated for a half-duplex method of operation is presented in Figure 4, which shows $\rho_{tot}(\tau_i, \eta_r)$ as a function of $\overline{E}_v$. Let Type I interactions be those in which whole pages of information are transmitted to the representative; parameter values assumed are $\overline{l}_r = 300$ characters and $\overline{I} = 3$ interactions. Let Type II interactions be those in which more specific items of information can be requested; values assumed are $\overline{l}_r = 75$ characters and $\overline{I} = 6$ interactions. Figure 4 indicates that in order to accommodate 30 erlangs, the less interactive method would require at least two half-duplex data links whereas one link may suffice for the Type II method, depending on the specified retrieval time objective.

## Volume of data traffic allowed per data link

For this example it has been assumed that the retrieval time objective would be stated as an average value, i.e., as $\overline{T}_{max}$. Mathematical queuing models have therefore been used for this example to aid in determining values of $\hat{\rho}_{tot}(\tau_i, \tau_r)$. Separate models were used to represent the half-duplex and full-duplex methods of operations; descriptions of these models and associated formulas are presented in the Appendix*; a derivation of the formulas for the half-duplex model appears in Reference 4.

* A computer simulation was used to verify the queuing model of the half-duplex method of operation. Values of $\overline{T}$ obtained with the queuing model were found to be conservative estimates. Additional discussion concerning the results of the queuing model and the simulation appears in the Appendix.

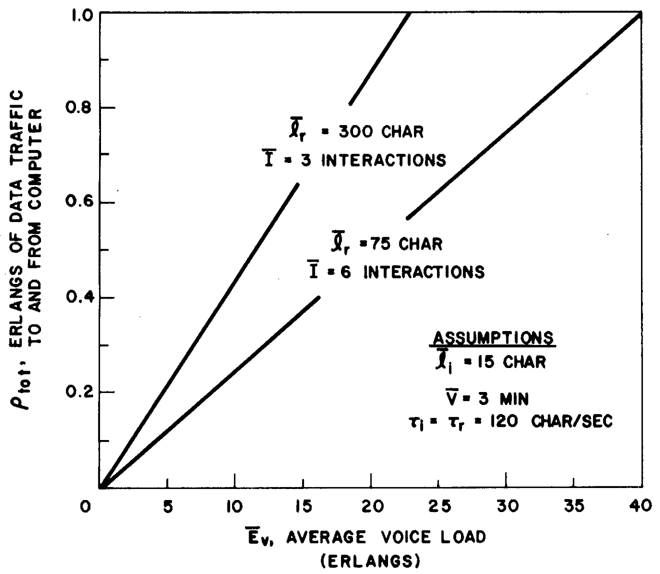


Figure 4—Conversion of voice load to data traffic

The correspondence between $\hat{\rho}_{tot}(\tau_i, \tau_r)$ and $\bar{T}_{max}$ was actually established in reverse, i.e., values of the average retrieval time $\bar{T}$ were calculated as a function of the total volume (in erlangs) of input and output data traffic carried by the data link. The five durations included in this retrieval time calculation are the following:

$D_i$ = Delay of an input message awaiting transmission

$t_i$ = Transmission time of the input message

$C_p$ = Computer processing time, i.e., interival from the arrival of the input message until the appropriate response is entered in an output queue

$D_r$ = Delay of the response in a computer output queue

$t_r$ = Transmission time of the complete response.

$\bar{T}$ was obtained by summing the mean value of these intervals, i.e.,

$$\bar{T} = \bar{D}_i + \bar{t}_i + \bar{C}_p + \bar{D}_r + \bar{t}_r \qquad (11)$$

The queuing models were used to determine values of $\bar{D}_i$ and $\bar{D}_r$. Since the server in these models represents the data link, these delay values depend on the average occupancy of the data link. For the half-duplex case, the average occupancy is numerically equal to $\rho_{tot}(\tau_i, \tau_r)$, providing $\rho_{tot}(\tau_i, \tau_r) < 1$. For the full-duplex case, the average occupancies of the input and output links are numerically equal to $\rho_i(\tau_i)$ and $\rho_r(\tau_r)$, respectively, providing $\rho_i(\tau_i)$ and $\rho_r(\tau_r)$ are both less than 1.

Values for $\bar{t}_i$ and $\bar{t}_r$ were obtained from Equations (5) and (6). The value of $\bar{C}_p$ was chosen to be two seconds; for other analyses, the value should be chosen to fit the characteristics and expected load of the system computer. With $\bar{C}_p = 0$, it should be noted that $\bar{T}$ represents the average retrieval time due solely to data communications, i.e., message queuing and message transmission.

In Figures 5A and 5B, $\bar{T}$ is plotted as a function of the erlangs of data carried per link for the half-duplex and the full-duplex cases, respectively; in each plot, $\bar{T}$ is shown for different average response lengths. For the half-duplex case, the erlangs of data carried per link is equivalent to the average data link occupancy. For each of the plots, as the erlangs of carried traffic approaches zero, $\bar{T}$ approaches the sum of $\bar{t}_i$, $\bar{t}_r$, $C_p$, and R, where R = 0.2 seconds for the half-duplex case and zero for the full-duplex case. Figure 5A can be converted into plots of $\bar{T}$ versus $\bar{E}_v$ by reference to Figure 4.

Other communications parameter values assumed for the plots in Figures 5A and 5B are as follows:

$$\bar{\ell}_i = 15 \text{ characters}$$
$$c^2(\ell_i) = 0.1^*$$
$$c^2(\ell_r) = 0.5$$
$$\bar{C}_p = 2 \text{ seconds}$$
$$\tau_i = \tau_r = 120 \text{ characters per second.}$$

The queuing models permit values of each of these parameters to be varied individually or in various combinations; by observing the results of such variations, insight is gained concerning which parameters most significantly affect $\bar{T}$. As was mentioned previously, the graphs can also be used in reverse to determine the effect of parameter variation on values of $\hat{\rho}_{tot}(\tau_i, \tau_r)$ for specified values of $\bar{T}_{max}$.

### Number of customer representatives allowed per data link

By relating values of S and $\rho_{tot}(\tau_i, \tau_r)$ appearing in Figures 3 and 4, respectively, that correspond to equal

* The coefficient of variation of a random variable y, which is denoted by $c^2$ (y), is defined as follows: $c^2(y) = \mathrm{Var}(y)/\bar{y}^2$.

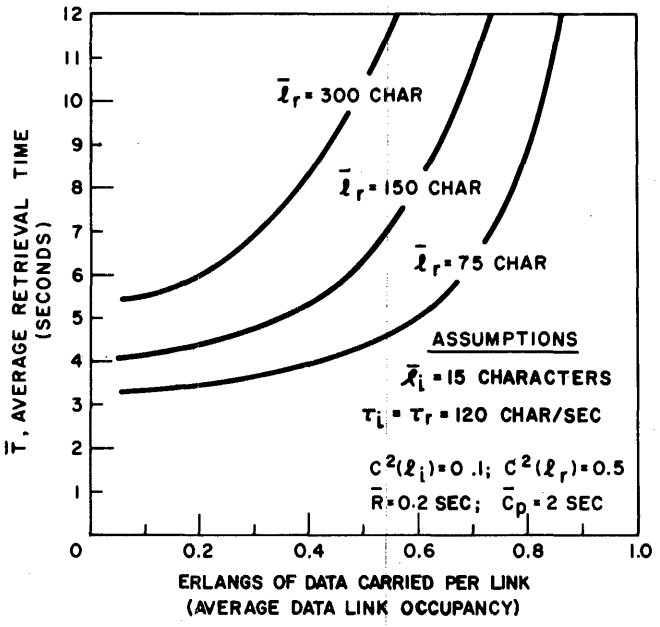


Figure 5A—Effect of response length on average retrieval time (Half-duplex message transmission)

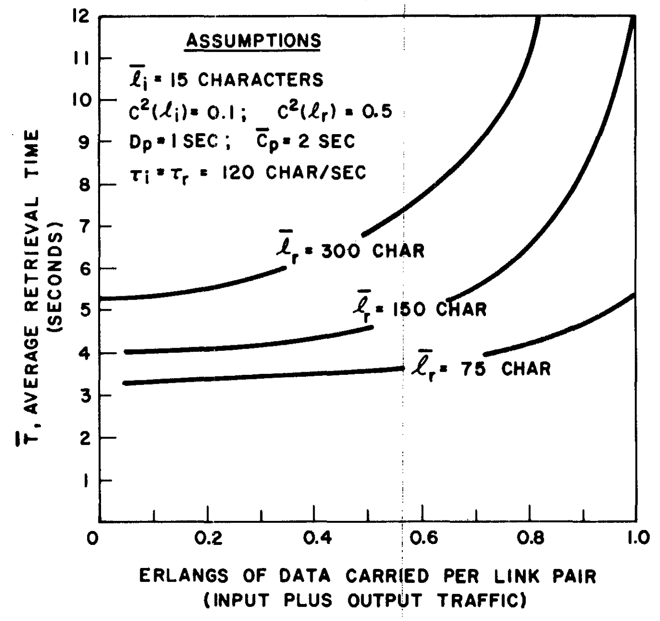


Figure 5B—Effect of response length on average retrieval time (Full-duplex message transmission)

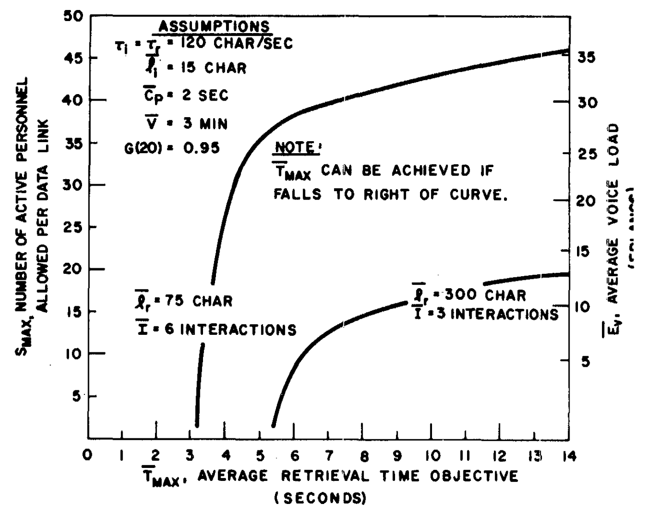


Figure 6A—Effect of response length on personnel allowed per data link (Half-duplex message transmission)

values of $\bar{E}_v$, a graph of S versus $\rho_{tot}(\tau_i, \tau_r)$ was obtained. This graph was then used in conjunction with Figure 5A, which shows T versus $\rho_{tot}(\tau_i, \tau_r)$ in order to obtain Figure 6A, which shows $S_{max}$ versus $\bar{T}_{max}$ for the half-duplex method of operation. Values of $\bar{E}_v$ that correspond to values of $S_{max}$ for G(20) = 0.95 are indicated on the right-hand vertical boundary of the graph. Curves are plotted to depict the effect of Type I and Type II interactions. Values of $\bar{T}_{max}$ that fall to the right of these curves can be achieved.

The graph indicates that for the indicated parameter values, the more interactive procedure allows considerably more personnel to be served by a single data link. With Type I interaction, i.e., the less interactive procedures, a $\bar{T}_{max}$ of five seconds cannot be met. However, with Type II interactions, this objective can be met for values of S less than approximately 35 representatives, which would be required to handle an incoming voice load of approximately 26 erlangs. If a $\bar{T}_{max}$ of three seconds is desired, it is obvious that some of the parameter values must be changed. Perhaps the transmission rates $\tau_i = \tau_r$ could be increased, or if possible, $\bar{C}_p$ could be reduced. Trade-offs can thus be studied between data communications and computer processing capabilities.

Figure 6B shows $S_{max}$ versus $\bar{T}_{max}$ for full-duplex message transmission. As expected, the graph indicates that full-duplex operation allows more representatives to be served on one data link than does half-duplex operation. Figure 6B also reveals that with full-duplex message transmission, a $\bar{T}_{max}$ of five or possibly four seconds can be met with one data link while handling 30 erlangs of incoming voice traffic. In comparison, reference to Figure 6A reveals that with half-duplex operation, two data links would be required to meet a $\bar{T}_{max}$ of five seconds with $\bar{E}_v$ = 30 erlangs; with one data link, $\bar{T}$ would equal approximately seven seconds.

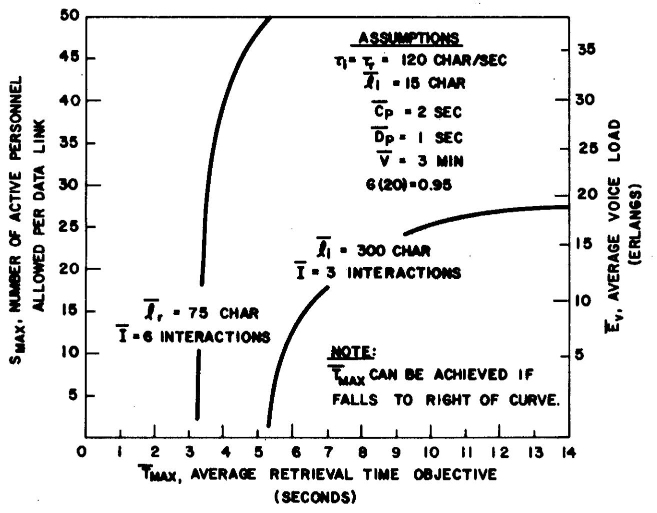


Figure 6B—Effect of response length on personnel allowed per data link (Full-duplex message transmission)

## SUMMARY

An analytical model has been presented that can be used for preliminary investigations of the voice and data communications aspects of inquiry-response systems. The model can be used to gain insight and to obtain conservative estimates of communications capabilities required in order for a system to meet specified grade-of-service objectives.

In particular, the mathematical relationships in the model can be used to estimate quantities such as the number o customer representatives required to handle incoming voice traffic and the volume of data traffic generated as a result of this voice traffic. These estimates in conjunction with retrieval time estimates are used to predict the number of data links required and the number of equivalently active input-output terminals that can be served by a data link without exceeding a specified retrieval time objective.

The model is useful for studying the sensitivity of the voice and data communications requirements to changes in various communications parameter values. This insight can aid in limiting the cost of subsequent detailed system simulations. Also, the model can be used iteratively to determine which combinations of parameter values permit specified voice traffic and retrieval time objectives to be met most economically.

As an illustration, the model is applied to a hypothetical system. Requirements for full-duplex and half-duplex message transmission are compared. The assumed methods of operation are characterized by group polling of and delivery to a common control unit rather than individual input-output terminals. For this application, estimates of average retrieval time as a function of erlangs of input and output data traffic were obtained by using delay formulas from mathematical queuing models.

## ACKNOWLEDGMENT

Mr. E. J. Rodriguez wrote and ran the computer simulation referred to in this paper. His assistance in obtaining this data as well as other data appearing in the figures is much appreciated.

## REFERENCES

1 A DESCLOUX
*Delay tables for finite- and infinite-source systems*
McGraw-Hill Book Company Inc N Y 1962 4

2 E C MOLINA
*Application of the theory of probability to telephone trunking problems*
Bell System Tech Journal Vol 6 1927 461-494

3 D MEISTER D E FARR
*The utilization of human factors information by designers*
Human Factors Vol 10 1967 71-87

4 J S SYKES
*Analytical model of half-duplex interconnections of computers*
IEEE Trans on Com Tech Vol 17 1969 235-238

5 *Analysis of some queuing models in real-time systems*
IBM Tech Pub Dept F20-0007-1 16

## APPENDIX

Queuing models were used in the application section of this paper to represent the assumed methods of operation of the data communications subsystem. This appendix contains a description of these queuing models as well as the associated delay formulas used for calculating $\overline{D}_i$ and $\overline{D}_r$, two of the terms in the expression for $\overline{T}$.

### *Half-duplex message transmission*

The queuing model selected to represent the assumed half-duplex method of operation is a single-server dual-queue model[4] in which service is alternated between the two queues; a finite interval is required to switch service from one queue to the other. Each queue is assumed to have an independent Poisson input and an independent general service time distribution. The alternating priority rule is followed. With this rule, all customers entering a queue while that queue is being served are also served; when that queue eventually becomes empty, service can be switched to the other queue.

In this model the single-server represents the data link that alternately allows transmission of the input messages that accumulate in the control unit and the responses that accumulate in the computer. The service times in the model represent the intervals $t_i$ and $t_r$ required to transmit individual messages. The switching, or reversal, times represent the intervals required to reverse the direction of data link transmission. For calculation purposes, it can be assumed that the reversal times also include the constant intervals required to transmit a fixed number of characters for supervisory purposes. Examples are polling sequences to request input messages from a control unit and identification sequences that precede input messages to identify the transmitting control unit.

Assuming the transmission times $t_i$ and $t_r$ have mean values $\bar{t}_i$ and $\bar{t}_r$ and coefficients of variation $c^2(t_i)$ and $c^2(t_r)$ and assuming the facility reversal time R, the polling time $P_1$, and the control unit identification time $P_2$ have constant durations, the formula for $\overline{D}_i$ is as follows:

$$\overline{D}_i = \frac{\rho_i\bar{t}_i g_i}{2(1-\rho_i)} + \frac{\rho_r g_r \bar{t}_r (1-\rho_i)^2 + \rho_i g_i \bar{t}_i \rho_r^2}{2(1-\rho_i)(1-\rho)(1-\rho+2\rho_i\rho_r)} + \frac{(1-\rho_i)(J_1+J_2)}{2(1-\rho)}$$

where

$$\begin{aligned}
\rho &= \text{average occupancy of the data link}\\
&= \rho_i + \rho_r < 1\\
\rho_i &= \lambda_i\bar{t}_i < 1\\
\rho_r &= \lambda_r\bar{t}_r < 1\\
J_1 &= (R + P_1)\\
J_2 &= (R + P_2)\\
g_i &= [1 + c^2(t_i)]\\
g_r &= [1 + c^2(t_r)]
\end{aligned}$$

Note that as $\rho \rightarrow 0$,

$$\overline{D}_i \rightarrow \frac{(J_1 + J_2)}{2}$$

The formula for $\overline{D}_r$ is identical to the one shown for $\overline{D}_i$ with all i subscripts changed to r's and vice versa.

Two additional formulas that may be helpful in estimating storage usage at the control unit and at the computer are the following, which give the average number of input messages and responses, respectively, that would be included in a multimessage transmission:

$$\overline{N}_i = \frac{\lambda_i[J_1 + J_2]}{(1-\rho)}$$

$$\overline{N}_r = \frac{\lambda_r[J_1 + J_2]}{(1-\rho)}$$

A computer simulation was used to determine how well this queuing model represents the assumed method of operation. Values of T obtained with the queuing model were found to be conservative. In general, the best agreement was obtained as long as values of $\rho$ were less than 0.6 to 0.7; differences were within a range from zero to 15 percent. With most combinations of parameter values, the disparity increased significantly for values of $\rho$ exceeding 0.8; i.e., the queuing model gave overly conservative estimates of $\overline{T}$. Agreement improved as the value of $\alpha = \bar{\ell}_r/\bar{\ell}_i$ decreased and/or the value of $\tau_i = \tau_r$ increased.

The disparity can be explained as follows: in the simulation the arrival pattern of responses in the computer output queue was not quite as random as is expected for Poisson arrivals, which are assumed in the queuing model. A principle of queuing theory is that as regularity of arrivals and service times increase, the average delay decreases.[5] Excellent agreement between the results of the queuing model and the simulation were achieved when the value of $\lambda_r$ used in the queuing model was set equal to 0.9 times the $\lambda_r$ used for the simulation.

*Full-duplex message transmission*

Independent models were selected to represent the input message queue and the response queue in the assumed full-duplex method of operation. Polling interference on the delivery line was assumed to be negligible.

For the response queue, the classical M/G/1 model was assumed. For the input message queue, an accumulation interval of $D_p$ seconds was assumed prior to each poll. This situation was modeled as an M/G/1 queue with a setup time of $D_p$. Assumptions for $t_i$ and $t_r$ are the same as stated for the half-duplex case. Formulas for

$D_i$ and $D_r$ are as follows:

$$\overline{D}_i = \frac{\lambda_i \overline{t_i^2}[1 + c^2(t_i)]}{2(1 - \rho_i)} + \frac{D_p}{2}$$

$$\overline{D}_r = \frac{\lambda_r \overline{t_r^2}[1 + c^2(t_r)]}{2(1 - \rho_r)}$$

Expressions for $\overline{N}_i$ and $\overline{N}_r$ for this case are as follows:

$$\overline{N}_i = \frac{\lambda_i D_p}{(1 - \rho_i)}; \quad D_p > 0$$

$$\overline{N}_r = \frac{1}{1 - \rho_r}.$$

# A study of asynchronous time division multiplexing for time-sharing computer systems

*by* W. W. CHU*

*Bell Telephone Laboratories, Incorporated*
Holmdel, New Jersey

## INTRODUCTION

In order to reduce the communications costs in time-sharing systems and multicomputer communication systems, multiplexing techniques have been introduced to increase channel utilization. A commonly used technique is Synchronous Time Division Multiplexing (STDM). In Synchronous Time Division Multiplexing, for example, consider the transmission of messages from terminals to computer, each terminal is assigned a fixed time duration. After one user's time duration has elapsed, the channel is switched to another user. With synchronous operation, buffering is limited to one character per user line, and addressing is usually not required. The STDM technique, however, has certain disadvantages. As shown in Figure 1, it is inefficient in capacity and cost to permanently assign a segment of bandwidth that is utilized only for a portion of the time. A more flexible system that efficiently uses the transmission facility on an "instantaneous time-shared" basis could be used instead. The objective would be to switch from one user to another user whenever the one user is idle, and to asynchronously time multiplex the data. With such an arrangement, each user would be granted access to the channel only when he has a message to transmit. This is known as an Asynchronous Time Division Multiplexing System (ATDM). A segment of a typical ATDM data stream is shown in Figure 2. The crucial attributes of such a multiplexing technique are:

1. An address is required for each transmitted message, and
2. Buffering is required to handle the random message arrivals.**

If the buffer is empty during a transmission interval, the channel will be idle for this interval.

An operating example of an ATDM system for analog speech is the "Time Assignment Speech Interpolation" (TASI) system used by the Bell System on the Atlantic Ocean Cable.[1] Using TASI, the effective transmission capacity has been doubled and the system operates with a negligible (with respect to voice transmission) overflow probability of about 0.5 percent, even without buffering.

The feasibility of the ATDM system depends on: (1) An acceptably low overflow probability—of the same or lower order of magnitude as the line error rate—that can be achieved by a reasonable buffer size, and (2) an acceptable expected message queuing delay due to buffering. To estimate these parameters, analyses of the statistical behavior of the buffer are presented below. The user-to-computer traffic is in

* Present address: Computer Science Dept., UCLA, Los Angeles California, 90024.

** There may be other reasons for providing buffering such as: tolerating momentary loss of signals (e.g., fading), momentary interruptions of data flow, permitting error control on the line, etc. Under these conditions, the buffer should be designed to satisfy also the above specific requirements.

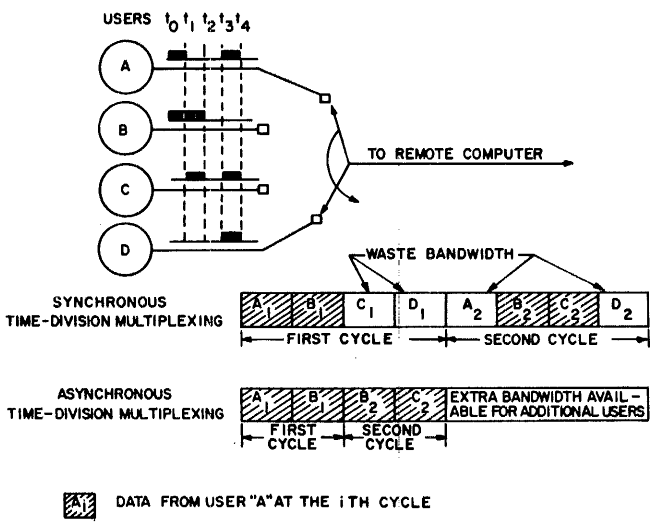

USERS $t_0 t_1 t_2 t_3 t_4$
A
B
C
D
TO REMOTE COMPUTER
WASTE BANDWIDTH
SYNCHRONOUS TIME-DIVISION MULTIPLEXING
FIRST CYCLE
SECOND CYCLE
ASYNCHRONOUS TIME-DIVISION MULTIPLEXING
EXTRA BANDWIDTH AVAILABLE FOR ADDITIONAL USERS
DATA FROM USER "A" AT THE i TH CYCLE

Figure 1—Time-division multiplexing

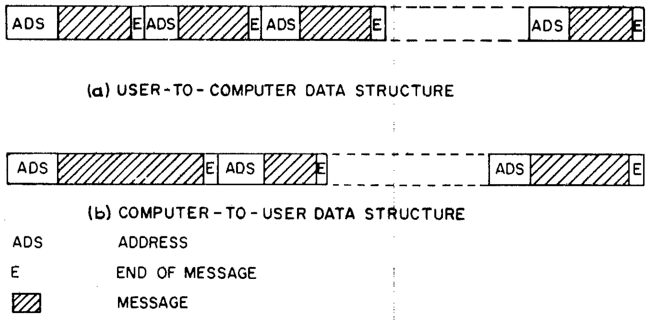

ADS
E
(a) USER-TO-COMPUTER DATA STRUCTURE
(b) COMPUTER-TO-USER DATA STRUCTURE
ADS ADDRESS
E END OF MESSAGE
MESSAGE

Figure 2—Asynchronous time division multiplexing data stream

units characters, while the computer-to-user traffic is in units strings of characters which we shall call bursts. The length of the bursts are different from one to another and are treated as random variables. Because of the asymmetrical nature of the traffic characteristics, the statistical behavior of the buffer in the user-to-computer multiplexer and the computer-to-user multiplexer are quite different and, therefore, are treated separately. An example is given to illustrate the multiplexer design in a time-shared computer-communications system that employs ATDM technique.

*Analysis of buffer behavior*

## User-to-computer buffer

An ATDM system consists of a buffer, encoding/decoding circuit, and a switching circuit (in the case of multiple multiplexed lines) as shown in Figure 3. For the analysis of the statistical behavior of user-to-computer buffer, the character (fixed length) arrivals

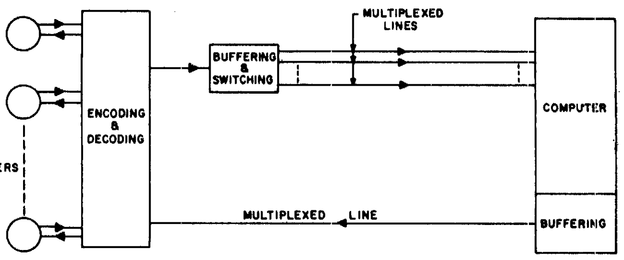


Figure 3—Asynchronous time division multiplexing system for time-sharing computer communications

from the sources to the buffer are assumed to be generated from a renewal counting process; that is, the character interarrival times are independent and identically distributed. Since the line transmits with constant speed, the time it takes to transmit each fixed length character (service time), $1/\mu$, is assumed to be constant. For reliability and simplicity in data transmission, synchronous transmission is assumed. The data are taken out synchronously from the buffer for transmission at each discrete clock time. The data arriving at the buffer during the periods between clock times have to wait to begin transmission at the beginning of the next clock time, even if the transmission facility is idle at the time of arrival. In queuing theory terminology, the above system implies there is a gate between the server and waiting room which is opened at fixed intervals. Thus we shall analyze the queuing model† with finite buffer size (waiting line) and synchronous multiple transmission channels (servers). Powell and Avi-Itzhak[2] analyzed a similar queuing model with an unlimited waiting line. Birdsall,[3] and later Dor[4] analyzed a queuing model with limited waiting room but with a single server. In here, the model is generalized to accommodate multiple servers with limited waiting room.

To establish the set of state equations for analysis of a buffer with a size of N characters and c servers, we assume that the system has reached its equilibrium. Let $p_n$ be the probability that there are exactly n characters in the system (in the buffer and in service) at the end of a service time, and $a_c$ be the probability

† The results derived from this study can also be used as a conservative estimate (upper bound) for the case in which the lines are permitted to transmit the characters arrived during the service interval. The estimate yields better approximation for the heavy than light traffic intensity case. Because under heavy traffic case, the lines are usually all busy and the characters that arrive during the service interval have to wait and cannot be serviced during the service interval. The maximum over design in a buffer system with c transmission lines that permits to transmit the characters arrived during service interval is c characters.

there are no more than c characters in the system at that time, i.e.,

$$a_c = \sum_{i=0}^{c} p_i \tag{1}$$

Without loss of generality, we can let the service interval equal to unity. We shall express the probability of number of characters present in the buffer at the end of the unit service time interval (left side of equation (2)) in terms of the probability of the number present in the system at the beginning of the interval (right side of equation (2)), multiplied by the probability of a given number of characters arriving during the service interval. As this can occur in different combinations, we add the probabilities. With synchronous transmission, all characters in service would finish their service and leave this system at the end of a service interval.

Thus in a unit service interval of time, we have

$$\left.\begin{aligned}
p_0 &= a_c\pi_0 \\
p_1 &= a_c\pi_1 + p_{c+1}\pi_0 \\
p_2 &= a_c\pi_2 + p_{c+1}\pi_1 + p_{c+2}\pi_0 \\
&\cdot \qquad \cdot \\
&\cdot \qquad \cdot \\
&\cdot \qquad \cdot \\
p_n &= a_c\pi_n + p_{c+1}\pi_{n-1} + \cdots + p_{c+n-1}\pi_1 \\
&\qquad + p_{c+n}\pi_0, \text{ for } n \le N - c \\
&\cdot \qquad \cdot \\
&\cdot \qquad \cdot \\
&\cdot \qquad \cdot \\
p_n &= a_c\pi_n + p_{c+1}\pi_{n-1} + \cdots \\
&\qquad + p_{N-1}\pi_{n+1-(N-c)} + p_N\pi_{n-(N-c)} \\
&\qquad \text{for } N - 1 \ge n > N - c \\
\sum_{i=0}^{N} p_i &= 1
\end{aligned}\right\} \tag{2}$$

Due to limited buffer size,

$$p_{i>N} = 0 \tag{3}$$

Where

$\pi_n$ = probability of n characters originating from a renewal counting process during a service interval

N = buffer length in characters

c = number of transmission lines

The first equation describes the case in which the buffer is vacant, if no more than c characters are in transmission at the beginning of the interval, and no arrivals occur during the interval. The second equation describes the case in which one character is in the buffer if no more than c characters are in transmission at the beginning and one arrives during the service time interval; or there are c + 1 in the buffer at the beginning and no character arrives during the service interval, etc. In the numerical computation carried out in this paper, we assume the character arrivals are generated from a Poisson process; that is, $\pi_n = \exp(-\lambda_u)\lambda_u^n/n!$, where $\lambda_u$ is the average character arrival rate to the user-to-computer buffer (offered load) from the m independent users. Since the buffer has a finite size of N, $p_{i>N} = 0$. Thus, when a character arrives and finds the buffer is full, an overflow will result. Therefore, the average character departure rate from the user-to-computer buffer (carried load), $\alpha_u$ is less than the offered load from the users $\lambda_u$. The carried load can be computed from the buffer busy period

$$\alpha_u = \sum_{i=0}^{c-1} i\cdot p_i + c\sum_{i=c}^{N} p_i \tag{4}$$

The overflow probability of the user-to-computer buffer, the expected fraction of total number of characters rejected by the buffer, is then equal to

$$P_{of} = \frac{\text{offered load—carried load}}{\text{offered load}} = 1 - \alpha_u/\lambda_u \tag{5}$$

The traffic intensity from user-to-computer, $\rho_u$, measures the degree of congestion and indicates the impact of a traffic stream upon the service streams. It is defined as

$$\rho_u = \lambda_u/c\mu \tag{6}$$

Channel (server) utilization, $\eta$, measures the fraction of time that the lines are busy. It can be expressed as

$$\eta = (1 - P_{of})\lambda_u/c\mu = \alpha_u/c\mu \le \rho_u \tag{7}$$

Since physically it is impossible for the transmission lines to be more than 100 percent busy, the utilization is limited to a numerical value less than unity. In the no-loss case (unlimited buffer size), $P_{of} = 0$, then $\eta \equiv \rho$.

The time average queuing length in the user-to-computer buffer, $L_u$, is equal to

$$L_u = \sum_{i=c}^{N} (i - c)p_i + \lambda_u/2 \text{ characters} \quad \text{for } N > c. \quad (8)$$

The first term in Equation (8) is the expected number of characters in the system at the beginning of a service interval. Since the characters could not leave the system during the service interval, we add the time average number of character arrival (for Poisson arrivals) during the service interval which is $\lambda_u/2$. The expected (time average) queuing delay of each character at the user-to-computer buffer due to buffering, $D_u$, can be evaluated by using Little's[5] result. We have

$$D_u = L_u/(\lambda_u(1 - P_{of})) \text{ service times} \quad (9)$$

For the single server case, that is, c = 1, the set of state equations (2) becomes an imbedded Markov Chain, and can be solved iteratively to obtain the state probabilities as shown in References 3 and 4. For the multiple server case, however, the multiple dependence on the various states prevents us from using the iterative techniques for solution. Thus, the set of state probabilities, $p_i$'s, must be solved from the set of linear matrix equations (2). The overflow probability, queuing delay, and queue length are then computed from the $p_i$'s via Equations 4, 5, 8 and 9.

The size of the matrix (Equation 2) corresponds to the buffer length. The matrix equation was solved by the Gauss elimination method.[6] For purposes of accuracy, double precision was used in all phases of the computation. From the character arrival rate, $\lambda_u$, the coefficient values can be computed from (2) and they are stored in the computer program. Due to the limitation of the computer word size, double precision on IBM 360/65 provides 15-digit accuracy. Therefore, when the coefficient value is less than $10^{-15}$, it is set equal to zero. The computation time required to solve this type of system equation is largely dependent on its size. For a 10×10 matrix the computation time is about 0.8 seconds, while a 50×50 matrix equation takes about 1.67 minutes.

Numerical results are presented in Figures 4, 5 and 6. These results reveal the relationships among the overflow probabilities, number of transmission lines used, traffic intensities, and buffer sizes.

## Computer-to-user buffer

In a previous section, the buffer behavior has been analyzed for a finite queue with multiple server, Poisson arrivals, and constant service time, which corresponds to the users-to-computer traffic. The

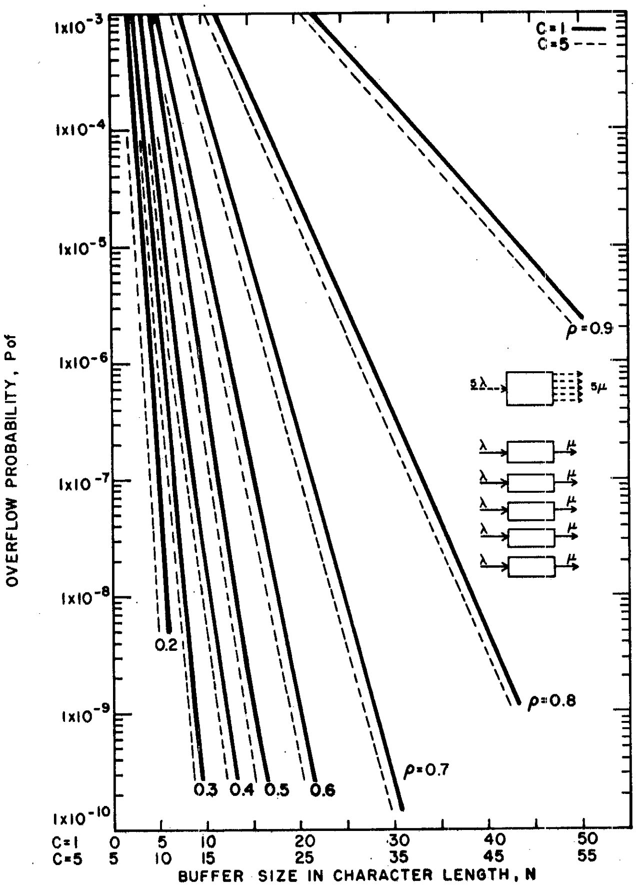


Figure 4—Overflow probability vs buffer size

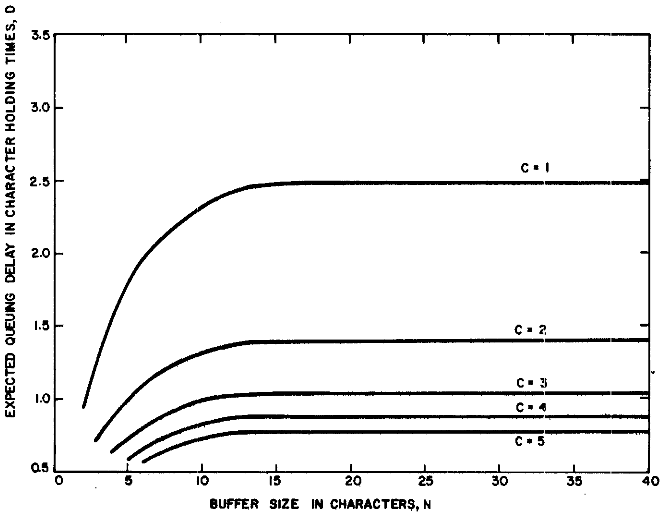


Figure 5—Expected queuing delay vs buffer size

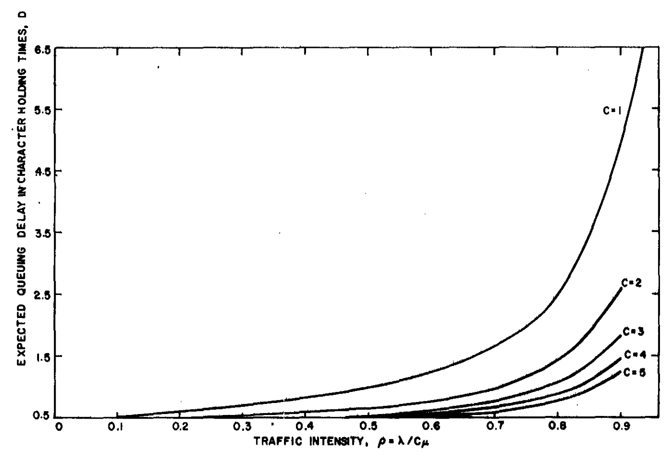


Figure 6—Expected queuing delay vs traffic intensity

computer-to-user traffic, however, is quite different from the users-to-computer traffic. The central processor of a time-sharing computer sequentially performs fractions of each user's job and the output traffic to the users are strings of characters which we shall call bursts. The length of the bursts are different from one to another and are treated as random variables. It is assumed that the internal processing speed of the computer is very fast as compared to the line transmission speed. Further, it is assumed that the various processing tasks generated by the user-computer interactions are independent from one user to another and have exponential interarrival times for a given user. In ATDM operation with these assumptions, the arrivals of bursts at the common output transmission buffer for the group of users are approximated as random. In this section, we shall analyze this buffer behavior under the assumptions of a finite queue, single server with batch (burst) arrivals, and constant service time.

Using the burst length and traffic intensity as parameters, we would like to find the relationships among the overflow probabilities, expected burst delays due to buffering, and buffer sizes.

Let us consider the case that the burst length, $L$ is geometrically distributed with mean, $\bar{\ell} = 1/\theta$; and the number of bursts arrived during a unit service interval (time to transmit a character from the multiplexed line), $N$, is Poisson distributed with mean, $\lambda_c$ bursts/service time. The distributions of $L$ and $N$ are as follows:

$$f_L(\ell) = \theta(1-\theta)^{\ell-1} \qquad \ell = 1, 2, \cdots \tag{10}$$

$$f_N(n) = \exp(-\lambda_c)\lambda_c^n/n! \qquad n = 0, 1, 2, \cdots \tag{11}$$

The total number of characters that arrived during the time to transmit a character on the multiplexed line is a random sum, $S_N$, and is equal to

$$S_N = \sum_{i=0}^{N} L_i \tag{12}$$

where $L_i$, a random variable distributed as (10), is the number of characters contained in the ith arriving burst. $N$, a random variable distributed as (11), is the total number of bursts arriving during the unit service interval. For simplicity in notation, we let $S = S_N$.

The characteristic function of S, $\phi_S(u)$, can be expressed in terms of the characteristic function of $L$, $\phi_\ell(u)$, and $\lambda_c$.

$$\phi_S(u) = \exp[-\lambda_c + \lambda_c\phi_L(u)] \tag{13}$$

Since the burst lengths are geometrically distributed the characteristic function of $L$ is

$$\phi_L(u) = \theta\cdot\exp(iu)/\Big(1-(1-\theta)\exp(iu)\Big) \tag{14}$$

where $i = \sqrt{-1}$. Substituting (14) into (13), then

$$\phi_S(u) = \exp[-\lambda_c + \lambda_c\cdot\theta\cdot\exp(iu)/ (1-(1-\theta)\exp(iu))] \tag{15}$$

From (15), it can be shown that the probability density of j characters arriving during a unit service interval, $f(S = j) = f_j$, is a compound Poisson distribution as shown in (16)

$$f_j = f(S=j) = \begin{cases} \displaystyle\sum_{k=1}^{j} \binom{j-1}{k-1} (\lambda_c\theta)^k (1-\theta)^{j-k}\exp(-\lambda_c)/k! & j = 1, 2, \cdots \\ \exp(-\lambda_c) & j = 0 \end{cases} \tag{16}$$

The expected value of S is given by $E[S] = E[L]E[N] = \lambda/\theta$, and the variance of S is given by

$$\mathrm{Var}[S] = \lambda(2-\theta)/\theta^2 \tag{17}$$

The time required to compute the probability density function of S, $f_j$, from (16) is dependent on the size of j. For large j (e.g., j > 1000), the computation time could be very large and prohibitive. A convenient and less time consuming way to compute $f_j$ is from $\phi_S(u)$ by using the Fast Fourier Transform[7] inversion method as follows:

$$f_j = \sum_{r=1}^{M} \phi_S(r)\exp[-2\pi irj/M]$$

$$j = 0, 1, 2, \cdots, M - 1 \qquad (18)$$

where

$r = 2\pi u/M$

M = total number of input points to represent $\phi_S(r)$ = total number of output values of $f_j$.

In order to accurately determine $\phi_S(r)$, it is computed with double precision on the IBM 360/65. Further, we would like to use as many points as possible to represent $\phi_S(r)$; that is, we would like to make M as large as possible. Because of the word length limitation of the computer, double precision provides 15-digit accuracy. Therefore, when $f_j < 10^{-15}$, it is set equal to zero. M is selected such that $f_{j \geq M} < 10^{-15}$. The M's are different for different values of $\lambda_c$ and $\bar{\ell}$.

The following is the set of state equations for a buffer size of N characters with batch renewal arrivals, single server, and constant output rate.

$$p_n = \pi_0 p_{n+1} + \sum_{i=1}^{n} \pi_{n-i+1} p_i + \pi_n p_0$$

or

$$p_{n+1} = \frac{1}{\pi_0}\left[p_n - \sum_{i=1}^{n} \pi_{n-i+1} p_i - \pi_n p_0\right] \qquad (19)$$

$$n = 0, 1, 2, \cdots, N - 1$$

$$\sum_{i=0}^{N} p_i = 1 \qquad (20)$$

and

$$p_{i>N} = 0 \qquad (21)$$

The above equations are reduced from Equation (2) by letting c = 1.

The average character departure rate from the buffer (carried load), $\alpha_c$, is less than the average character arrival rate to the buffer (offered load), $\beta = \lambda_c/\theta$, from the computer. The carried load can be computed from the probability that the buffer is idle,

$$\alpha_c = 1 - p_0 \qquad (22)$$

The overflow probability of the buffer with burst input, the expected fraction of total number of characters rejected by the buffer, is equal to

$$P'_{of} = \frac{\text{offered load-carried load}}{\text{offered load}} = 1 - \alpha_c/\beta \quad (23)$$

The traffic intensity from computer-to-user is

$$\rho_c = \beta/\mu = \lambda_c/(\theta\mu) = \lambda_c\bar{\ell}/\mu \qquad (24)$$

The set of state Equations (19) is an imbedded Markov Chain. In the following numerical computations, we shall assume that the character arrivals are generated from a compound Poisson process, i.e., $\pi_i = f_i$. The state probabilities can be solved iteratively and expressed in terms of $p_o$. From (20), we can find the value of $p_o$. Thus we find all the state probabilities. The overflow probabilities for various burst lengths can then be computed from (23). These results are presented in Figure 7 which provides the relationships (at $P'_{of} = 10^{-6}$) between burst lengths and buffer sizes for selected traffic intensities.

In the above analysis, we have treated each character as a unit. However, in computing the expected burst delay, $D_c$, due to buffering, we should treat each burst as a unit. The service time is now the time required to transmit the entire burst. For a line with

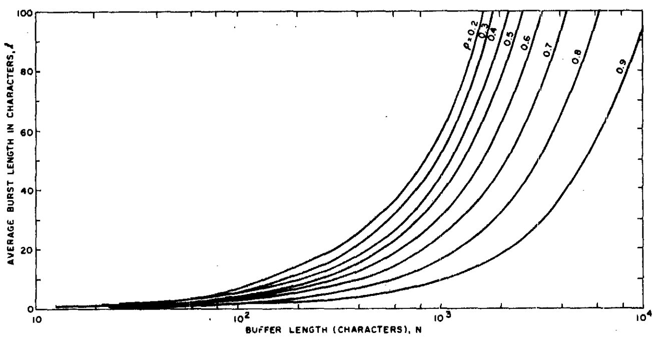


Figure 7—Buffer length vs average burst length, $P'_{of} = 10^{-6}$

constant transmission rate, the service time distribution is the same as the burst length distribution except by a constant transmission rate factor. When overflow probability is very small, for example, $P'_{of} = 10^{-6}$, then $D_c$ can be approximated by the expected burst delay of the infinite waiting room with Poisson Arrivals and single server with geometric service time, M/G/1, model.[8,9] Hence

$$D_c = \frac{\lambda E(L^2)}{2(1-\rho)} = \frac{\lambda_c(2-\theta)}{2(\theta-\lambda_c)}$$

character-holding times (25)

where $E(L^2)$ = second moment of burst length, $L$. The delays are computed from (25) for selected traffic intensities and burst lengths. Their results are portrayed in Figure 8.

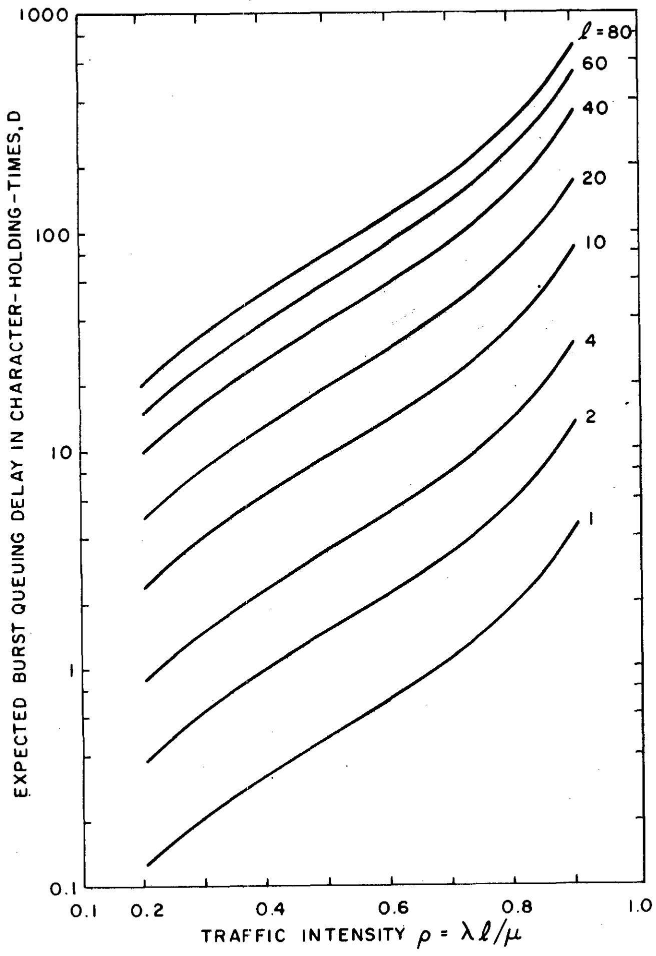


Figure 8—Traffic intensity vs expected burst queuing delay

*Discussion of results*

We shall first discuss the user-to-computer buffer behavior. Figure 4 portrays the relationships between overflow probabilities and buffer size for selected traffic intensities and selected numbers of servers. The curves for two-, three-, and four-servers lie in the region between the single and the five-server curves. For a given traffic intensity, the overflow probability decreases exponentially with buffer size. For a typical traffic intensity of 0.8 ,a buffer of twenty-eight character length will achieve an overflow probability in the order of $10^{-6}$. A larger buffer size is needed for $\rho_u > 0.8$ in order to achieve the same degree of buffer performance. For a given $\rho$, the queuing delay increases as the overflow probability decreases (or the buffer size increases). When the overflow probability is less than $10^{-4}$ (for $\rho_u$ = 0.8, this overflow probability corresponds to a buffer size of about eighteen characters), the delay increment with buffer length becomes negligible and the delay can be approximated as independent of buffer size as shown in Figure 5.

For the data transmissions in time-sharing systems, the buffer overflow probability should be somewhat less than the line error rate. For currently available lines, the error rate is about $10^{-5}$. Therefore from Figure 5, we know that the queuing delay range of interest is almost independent of the buffer length. Figure 6 describes the queuing delays (at overflow probability = $10^{-6}$) for various traffic intensities. The queuing delay increases exponentially with $\rho$. For a given $\rho$, the queuing delay decreases with the increase of number of servers. Figures 4 and 6 agree with our intuition that whenever multiple servers are needed, it is always advantageous to use a common buffer rather than using several single lines with separate buffers.

Next we shall discuss the computer-to-user buffer behavior. The overflow probability depends upon the buffer size, the traffic intensity, and expected burst length. For a given average buffer length, the overflow probability increases as the traffic intensity increases. For a given traffic intensity, and a desired buffer overflow probability, the required buffer size increases as the average burst length increases. Figure 7 provides the relationships between the average burst length and required buffer size to achieve an overflow probability of $10^{-6}$ for selected traffic intensities.

When the average burst length equals unity, then the result reduces to the case of Poisson arrivals, single server and constant service time as had been analyzed.[3,4] For a given traffic intensity, required buffer size for average burst lengths $\bar{\ell}(\bar{\ell} > 1)$, $N_{\bar{\ell}}$, to achieve the same degree of overflow probability is much greater than that for unity burst length, $N_1$. In general, $N_\ell > \bar{\ell} \times N_1$. As $\bar{\ell}$ increases, the difference between $N_\ell$ and $\bar{\ell} \times N_1$ increases. For example, for $\rho_c = .8$, $\bar{\ell} = 1$, the required buffer size to achieve $P'_{of} = 10^{-6}$ is $N_1 = 28$ characters. When $\bar{\ell} = 4$, then from Figure 7, $N_4 = 212 > 4 \times 28 = 112$ characters. In the same manner, if $\ell = 20$, $N_{20} = 1200 > 20 \times 28 = 580$ characters. This is due to the fact that the variance of S is proportional to $\bar{\ell}$ as shown in (17). Figure 8 portrays the relationship between expected burst queuing delay and traffic intensity for selected expected burst lengths. For a given expected burst length, the expected queuing delay increases as traffic intensity increases; for a given traffic intensity, the expected queuing delay increases with burst length. These are important factors that affect the delay.

*Optimal design of multiplexing system*

Let us first consider the design of the user-to-computer multiplexer. Based on the user-to-computer traffic characteristics, the number of user terminals, maximum allowable queuing delay, and overflow probability, several different buffer system configurations might satisfy the desired requirements. Hence there are trade-offs among the number of transmission lines we might use, the transmission rates of the lines, and the buffer sizes. We would like to design the multiplexing system whose total cost (transmission cost and buffer storage cost) is minimum. One way to proceed with this is first to select the set of possible multiplexing system configurations based on the queuing delay requirements from Figure 6. Based on the maximum allowable overflow probability, we can obtain the required buffer length for this set of possible multiplexing system configurations. The optimal user-to-computer part of the multiplexing system can then be selected as that which minimizes the cost of the system.

Next, we shall consider the optimizations of the computer-to-user multiplexer. Data collected from several operating time-sharing systems[10] revealed that the average number of characters sent by the computer to the group of users is an order of magnitude greater than the number of characters sent by the group of users to the computer. Thus, using high transmission rate line for computer output data would significantly reduce in buffer size and the queuing delay due to buffering. Further, the change in the computer system such as changes in the scheduling algorithm[11–17] in the central processor can strongly influence the computer output traffic statistics, which will directly affect the buffer performance, and the design of the decoding system.

In practice, we would like to design a system that has minimum total cost yet satisfies all the requirements such as the inquiry-response delay, average holding time of each user, etc. Since the multiplexing system and the central processor intimately interact with each other, the multiplexing system should be treated as a subsystem of the time-shared computer system. The economical and performance optimization should be carried out jointly between the central processor and available communication facilities.

*Example*

Consider the design of a time-sharing system that consists of many remote terminals and that employs the ATDM technique with full duplex operation between the terminals and the central processor. Measurements of the traffic characteristics from several operating systems have revealed that the character interarrival time per user line can be approximated as exponentially distributed with mean about 0.5 seconds.[10] Thus, the character arrivals can be treated as Poisson arrivals with a rate of 2 char/sec. A reasonable conservative guess is that 50 percent of the transmitted information is sufficient for addressing and framing. Voice-grade private lines can easily transmit 240 char/sec from users. Suppose this operating system consists of $m = 48$ terminals, all the terminals are assumed to be independent and have the same traffic characteristics. The buffer is designed such that the overflow probability is less than about $10^{-6}$. We shall use our model to determine the buffer size and the average queuing delay incurred by each character.

The traffic intensity is $\rho_u = 1.5 \times m\lambda_u / c\mu_u = 1.5 \times 48 \times 2/240 = 0.6$. To achieve the desired overflow probability, from Figure 4, the required buffer length is 14 characters. From Figure 6, the normalized queuing delay due to buffering is equal to 1.25 holding times. Since each holding time is equal to $1/\mu_u = 1/240 = 4.16$ millisecond, the waiting time of each character is 5.06 milliseconds. Now suppose the number of terminals is increased from 48 to 96. In order that traffic intensity be less than unity, two transmission lines are required and the traffic intensity is still equal to 0.6. From Figure 5, the buffer length corresponding to the desired overflow probability for two transmission lines is about 14 characters. The waiting time is about 0.8 holding times which is equal to 3.33 milliseconds. Although the difference between 5.06 milliseconds and 3.33 milliseconds may not be detected by a user at a terminal, a common buffer of the same size operating with two output lines can handle twice the number of input lines as with one output line. Thus, the common buffer approach permits handling a wide range of traffic without substantial variation in buffer size.

Next, we shall consider the buffer design problem that employs the ATDM technique to transmit data from central processor to remote terminals. The traffic statistics as well as the message length are different from that of the users. The burst interarrival time[10] can be approximated as exponentially distributed with a mean of 2.84 seconds. Thus, the bursts can be approximated as Poisson arrivals with a rate of $\lambda_c =$ 0.35 bursts/sec. Further, data collected in the same study indicate that the burst length can be approximated as geometrically distributed with a mean of $\bar{\ell}$ = 20 characters. Suppose we use a wideband transmission line that transmits 480 char/sec to provide communications from the central processor to 48 terminals. Assuming 20 percent of the transmitted information is used for addressing and framing, then the traffic intensity, $\rho_c = 1.2\lambda_c\bar{\ell},\mu_c \approx 0.84$. To achieve an overflow probability of $10^{-6}$, from Figure 7, we find that the required buffer size is 1,400 characters. From Figure 8, the expected queuing delay for each burst is 85 character-holding times, or $85/480 = 0.176$ seconds.

Suppose now we changed our transmission rate from 480 to 960 char/sec; then the traffic intensity $\rho_c \approx 0.42$. The corresponding required buffer size in order to achieve an overflow probability of $10^{-6}$ is 480 characters, and the delay is 15 character-holding times or 16 milliseconds. Thus, these results also provide insight regarding the trade-off between transmission costs and storage costs.

The above example is based on the output traffic characteristics of a specific computer scheduling algorithm. As the output traffic statistics changes with different scheduling algorithms, the buffer performance in the multiplexing system is affected. To design an optimal system, we should jointly optimize the scheduling algorithm and the multiplexing system such that yield minimum total cost and also meet the required system performance such as maximum allowable inquiry-response delay, desired overflow probability, etc.

## CONCLUSIONS

Queuing analyses indicate that for an allowable overflow probability and queuing delay, moderate buffer sizes can be achieved for asynchronous time division multiplexing for time-sharing computer systems. Further, when multiple transmission lines are required, better buffer performance will be achieved by using a common buffer rather than by using separate ones.

Because of the asymmetric nature of the traffic characteristics of user-to-computer transmission versus computer-to-user transmission, a much larger buffer is required for the computer-to-user multiplexer to handle the larger volume of data generated by the central processor.

The multiplexing system and the central processor in a time-shared environment directly interact with each other. To design an optimal operating system, we should jointly optimize the central processor and the multiplexing system (for example, the interaction between scheduling algorithm and buffer performance) to obtain a minimum cost system that meets the system performance requirements. It is apparent that closer coordination between the computer and communication system designs would be fruitful in terms of economics and technological improvements to the overall system design.

## ACKNOWLEDGMENTS

The author wishes to thank E. Fuchs and D. Heyman of Bell Telephone Laboratories for their helpful discussions.

## REFERENCES

1 K BULLINGTON J M FRASER
*Engineering aspects of TASI*
B S T J March 1959 353-364

2 B A POWELL B AVI-ITZHAK
*Queuing system with enforced idle time*
Operations Research Vol 15 No 6 Nov 1967 1145-1156

3 T G BIRDSALL et al
*Analysis of asynchronous time multiplexing of speech sources*
IRE Trans on Communications Systems Dec 1962 390-397

4 N M DOR
*Guide to the length of buffer storage required for random (Poisson) input and constant output rates*
IEEE Trans on E C Oct 1967 683-684

5 J D C LITTLE
*A proof of the queuing formula $L = \lambda W$*
Operations Research Vol 9 1961 383-387

6 R W HAMMING
*Numerical methods for scientists and engineers*
McGraw-Hill Book Co Inc N Y 1962 363-364

7 W M GENTLEMAN G SANDE
*Fast fourier transforms—for fun and profit*
Proc FJCC Vol 29 563-578

8 N U PRABHU
*Queues and inventories*
John Wiley and Sons Inc N Y 1965 42

9 P M MORSE
Queues Inventories and Maintenance

John Wiley and Sons Inc 1958 15-18

10 P E JACKSON C D STUBBS
*A study of multiaccess computer communications*
Proc SJCC Vol 34 1969 491-504

11 A L SCHERR
*An analysis of Time-Shared Computer Systems*
MIT Research Monograph No 36 MIT Press Cambridge Mass 1967

12 P E DENNING
*Effect of scheduling on file memory operations*
Proc SJCC Vol 30 1967 9-21

13 J E SHEMER
*Some mathematical considerations of time-sharing scheduling algorithms*
J ACM Vol 14 No 2 April 1967 262-272

14 E G COFFMAN JR
*Analysis of two time-sharing algorithms designed for limiting swapping*
J ACM July 1968

15 E G COFFMAN L KLEINROCK
*Feedback queuing models for time-shared system*
J ACM Vol 15 No 4 Oct 1968 549-576

16 L KLEINROCK
*Certain analytic results for time-shared processors*
Proc IFIP Congress 1968 Edinburgh Scotland Aug 5-10 1968 D119-D125

17 W W CHU
*Optimal file allocation in a multicomputer information system*
Proc IFIP Congress 1968 Edinburgh Scotland Aug 5-10 F80-85

# The involved generation—Computing people and the disadvantaged

*by* DAVID B. MAYER

*IBM Systems Development Division*
White Plains, New York

## INTRODUCTION

Motivated computer professionals all over the United States have undertaken a most special and extraordinary task: they are involving themselves in every way possible in the training of disadvantaged and educationally-deficited men and women from the so-called ghetto and poverty areas of the country. They are exhibiting a special and wonderful tension which impels them to appear at that interface between their own computing community and those underprivileged who wish to enter it.

As Chairman of the new ACM Committee On Computing And The Disadvantaged (ACM—CCD) I have been privileged to visit or directly participate in ten projects in New York City, Boston, Los Angeles, San Francisco, Sacramento, St. Louis, and Philadelphia. From them can be drawn some broad brush pictures of such projects, some of their special problems, and their relative probabilities of success.

### *The disadvantaged—Who are they?*

The term "disadvantaged" was originally coined in connection with educational grants from the government, for potentially very bright youths from proverty backgrounds for experiments in educational techniques programs. Since that time, it has broadened to include all those who are educationally-deficited (and with minimal hope of retrieval of those years they are behind), including those from both poor white and non-white communities.

Typically, computer projects have undertaken to train some of the disadvantaged either as operators or programmers. Generally the participants have been characterized as follows:

- 19-23 years old
- dropped out of ninth or tenth grade
- are black or brown
- are two to three years behind their white counterparts who are at the same grade level in terms of tested comprehension
- about two-thirds male
- have a job of some kind, but are underemployed apparently by reason of race or language
- come from a poverty-stricken area, often an urban "ghetto"
- have police records in about one-third the cases
- evidently have some motivation to better themselves
- have children or heavy "family" responsibilities
- on aptitude tests score over the complete range from high to low

More particularly though, a review of some other statistics may help us to orient ourselves:[17]

For Negroes in the 25-34 year old age bracket:

- 47.0 percent dropped out before graduation from high school
- 45.6 percent completed high school
- 7.4 percent completed high school and college
- A Negro sixth grader was 2-1/2 grade levels behind his white counterpart in general scholastic achievement
- A Negro ninth grader was three grade levels behind his white counterpart

This three-year deficit picture persists, through 12th grade and graduation, in general.

## Remediation, restructuring, and 'relevancy'

What does this mean to the computer training course, or to the jobs which people with such backgrounds can undertake?

It means some tutoring in the technical concepts during the computer operator's or other courses. It almost certainly will mean lengthening the course deliberately. Currently computer operator and programming (usually Cobol, by the way) courses run two to five times longer than the equivalent course given in the regular industrial milieu.

It means teaching only 'relevant' material, only the guts of content, only that which is directly applicable to that job waiting at the end of the course: ergo, no frills.

It means employers will have to restructure some jobs, in smaller, less complex, carefully detailed clusters, so that a rather straight-forward set of behaviors can be carried out by new employees.

It *is* possible to take small top level segments of the disadvantaged populace and train them directly in computer tasks without remediation. But in general if we want to really dig into the American dilemmas of today, remedial training will be needed for any broad training program developed to bring students up, to the level of comprehension needed to understand some of the computer concepts of our more abstruse computer texts.

## Trade-offs in training

There is, then, a kind of balance of course content requirements versus several variables—principally time—which one can invest to obtain effective training and eventual on-the-job performance results.

For example, most disadvantaged projects teaching key punch operators required that trainees be able to type 20 to 40 words per minute prior to entering key punch classes.[2,3,4] Where the normal key punch class is five days, in projects for the disadvantaged they run 15 to 20 days.

A project choosing high school *graduates* can train computer operators quite effectively, and include theoretical material on operating systems, programming techniques, the internal supervisor/program coupling within the computer, enough so as to allow an operator to make some reasoned judgements in error situations. This is obtained through trade-offs such as (a) lengthening the course, or (b) intensifying the hands-on expereince. This probably gives the disadvantaged person who graduates one of the finest running starts in 'operations' in the country. (N. B. Particularly true of the Urban League/IBM/Bank of America project in Los Angeles.[4]

### *The placement problem and the assumed job market*

Most projects have been located in large urban, highly computerized geographical areas; groups in the planning stages have typically looked about themselves and faced the combinatorial possibilities of probable jobs available and probable people they were hoping to train. Almost invariably they concluded that three possible combinations were feasible:

- key punch operator
- computer operator, either as a trainee handling tapes and discs and peripherals primarily, or as a trainee console operator.
- a trainee Cobol programmer

Generally *rejected* for training were job descriptions which involved:

- Fortran or basic language programmers
- pure EAM or "unit record equipment" operators; however, this was sometimes appended to the computer operator trainee position description
- tape librarians, dispatchers, I/O clerks, and the like.

Most projects made only a cursory pass at the actual placement planning question and generally assumed that any graduates they offered the marketplace would be snapped up with only a modicum of effort to find interviews. Inevitably, halfway through the training when efforts turned toward placement interviews there were some rather rude awakenings to several facts: the students' color, language, and prior records were obstacles that required active selling to overcome. More often than not there was a mad scramble toward the end of the training period to find employers willing and able to hire trainee computer operators from the poverty sector of our poplulation. Only heroic efforts upon the part of placement committees would slowly find openings for interviews, much less pre-committed employment slots.

Hence, if there were one piece of advice this author would give it would be: *plan your placement process first; involve would-be employers at the earliest planning stages* to test the marketplace, to involve them in the training stages, to be interested in the graduates, and to assure jobs at the end of the course. It is almost axiomatic that if you should fail to place your 'disadvantaged' trainee within a very few weeks of his graduation you may have lost him or her forever and all the training investment will have been for naught.

## *The computer operator—What is he?*

In order to converge upon concrete results and make some comparisons only the training surrounding the job of Trainee Computer Operator will be described.

Let us consider three different, but related, aspects of the Computer Operator position description:

- EAM or "unit record equipment" knowledge and/or ability;
- pure computer *operating*, highly *structured*, highly *practical*, based on detailed specified stimuli and response patterns.
- an *"understanding"* computer operator who has sufficient *theoretical* knowledge about operating systems computer operator, who has sufficient *theoretical* knowledge about operating systems to solve unexpected error situations, so as not to abort, but successfully run a job.

In a typical job description, the Trainee Computer Operator works under close supervision, performs the simpler operations on peripheral devices and on the console, expedites the data in and out of the system and the installation, and is generally a careful intelligent follower. He is usually expected to have two years of college (possibly an AA degree) or several years of tabulating machine (EAM) experience or a 200-hour hands-on computer operating course.[16]

The Journeyman Computer Operator is expected to do more: based upon six or more months of actual Trainee experience, he checks input and output for general results, analyzes stops and takes corrective action, and runs test programs. He is also required to know the principles of operations, basic elements of programming, follow directions carefully and analyze data, and perform arithmetic computations.

It is the author's contention that the EAM tasks and training are frills and basically obsolete and should not be taught (excepting a little keypunching for error corrections); that the second or "structured, practical" job description is the one for minimal entry level jobs for the disadvantaged; and that the third description adds a requirement for "theoretical understanding" for computer operator training projects. This latter requirement is significantly high in terms of language comprehension and acts as a deterrent to large sectors of the disadvantaged population trying to take advantage of the training.

It is interesting to note that in almost every case of the disadvantaged training projects with which the author is familiar, nowhere nearly such stiff conditions are placed either upon the students for entry into the computer operator course, nor upon them for eventual hire. And there is every indication they can perform successfully upon the job with considerably less stringent qualifications.[2,7,12]

The author therefore urges that to be able to develop the truly disadvantaged, educationally-deficited person (a dropout from as low as the ninth grade) computer installations should re-structure their basic computer operator job specifications and training projects their content to reflect the entry-level requirements for the "practical, structured" computer operator trainee. This would give gainful employment of a meaningful type to many more people in the total community, particularly the disadvantaged.

## *Computer operator curricula*

In this section are described two examples of operator curricula to exhibit the basic approaches, typical content, and a preliminary view of some of the training techniques employed. (Treated more fully in another section.)

In the Mitre Corporation's fully in-house, fully funded (internally) on-the-job (OJT) format[9] students are paid at regular industrial rates. Remedial training in basic language skills and mathematics takes up most training hours daily, for the first few of the 26 weeks total. Gradually it is replaced by computer operations training on both the IBM 7030 (Stretch) and the IBM 360/30 and 360/40 systems; and of course gradually the students work out on the line.

Instructors are internal, paid, staff members; four students started and three finished successfully. They were part of a 12-trainee Mitre project for clerks, operators, and the like.

The salient features of this Project's outline include: (a) deliberately assigning their second shift Supervisor for nine months as Training Coordinator to prepare the technical curriculum, instruct, supervise the OJT aspects, and coordinate with the remedial training; (b) giving all first shift personnel a stake in the outcome, and include them in the evaluation process; (c) providing separate remedial training on a descending scale concurrent with increasing line operations training and expereince. A full Outline is available from the ACM.[18,26]

The second basic approach and the one most often used, was the external, separate training program; it is typified by the CPDA project in New York City.[1] Using a "self-selection" process[12] 75 prospective students went through an 'orientation' to computer operating, and then 48 volunteered for actual training. Thirty-two stayed with it, 27 graduated on the first round, three more were tutored to completion, and 17 of the 30 were placed as of this writing.

The program, approach, and Syllabus Outline of CPDA are given in Figures 1 and 1a.

EAM/Unit Record equipment training is not given in this course. Several of the courses did offer as much as a week's equivalent of such training, on the basis of its relevance still in today's card-oriented input/output part of the computing world. The Urban League/IBM/Bank of America project in Los Angeles, the Philadelphia ACM/Board of Education project, and the St. Louis IBM/Board of Education are noted in particular .[4,10,11] The latter have a regular EAM course available in their vocational schools as well.

1. ORIENTATION PROGRAM

a. Registration

- Welcome 75 prospective students
- describe project
- history and needs of the computer field
- introduction to the computer
- film on computer and operations.

b. Introduction To Computer And Business Environment

- devices used at an installation
- business environment and general working conditions
- employment prospects for the computer operator.

c. Computer Installation Visit

d. The Computer Operator And The Training Program

- computer concepts
- general responsibilities of the operator on the job
- the operator's relationship to the computer field
- the training program
- general discussion and individual counseling.

Extensively, throughout these orientation sessions instructors and counsellors mingle with the students, interact on questions, and encourage *self*-selection into, or not-into, the actual training.

Figure 1—A no-frills training syllabus. Computer Operations Training for the IBM S/360 Models 30 and 40. Training consists of two parts. Since motivation and interest are prime factors in training completion and job performance, a preliminary four-session orientation program was designed to give the candidate a data base for making up his own mind, to enter or not-enter training. (After CPDA[1,18]).

*Training techniques*

It would seem obvious that some specialized training techniques would have to be employed to reach disadvantaged or educationally-deficited people, and a few such techniques have been attempted in the computer training field. Such experiments should be carried out in a professional, measured, feedback atmosphere, but rarely has that been available. In the projects studied, probably the three aspects of training that have paid off the most are:

- lengthening the courses by two to five times the average;
- making the classes small; or alternatively assigning two instructors per class to bring the pupil/teacher ratios down to as low as four-to-one;
- allowing the class to teach itself, to a certain extent, by teaming or as a full group.

Interestingly enough, no project used any specialized audio-visual material (other than hands-on work with the computer itself), most of them depending upon the standard available texts, programmed instruction books, or books of illustrations.

Nevertheless, a kind of experiment did take place in the CPDA project, observed by the teachers, staff professional guidance counselors, and the students themselves. It involved trying three differing teaching techniques:

1. The 'classical' approach consists of a teacher lecturing to his students, with the teacher as focus for feedback (answers, discussions, questions). This can be characterized as a 'vertical' organization of class structure.
2. The 'teams approach consisted of the teacher breaking up the group into five teams of three students each. This came about to solve the problem of demonstrating the computer console and peripherals effectively. Having 15 students stand around in a large semi-circle proved boring and ineffectual; by placing the few most hep students with two of each of the others, he could in effect assign problems to teams to work out, and allow students to teach each other within teams. When competition rather than cooperation started to raise its head, the team members were rotated. In addition, the instructors, after giving the teams a problem, deliberately gave the impression that they would answer no further questions. After computer runs the whole class would hold a post-mortem. Instructors also created unexpected problems, such as casually dereadying a printer, or flipping a tape into 'file protect' mode without a file protect ring being inserted. Furthermore, students would be called upon at random at the beginning of a class to recapitulate the pre-

2. COMPUTER OPERATIONS TRAINING PROGRAM

a. Course Structure

- The training program will consist of both classroom sessions and computer room visits.
- It is expected that approximately 30-45 students will complete the Orientation Program and enter the Operations Training Program.
- There will be three sections, each with 10-15 students.
- Each section will have one primary teacher and one assistant teacher.
- Classroom sessions will meet twice a week for two hours.
- Computer room visits will be scheduled as required by the Syllabus and will be from two to three hours in length.

b. Educational Material. The basic student text for the course will be:

IBM System 360 Model 30 DOS System Operation Training Manual and Book of Illustrations (Student Text); Forms C20-1676-0, C20-1677-0).

- Examples of I/O media will be available in the classroom for student familiarity with cards, tapes, disk packs, printer forms, carriage control tapes, etc.

SYLLABUS

Section A (INTRODUCTION AND PERIPHERALS) covers:

- Introduction to Input/Output Media
- Computer Room Procedures
- Computer Room Visits (hands-off demonstrations and hands-on practicums)
- Operations of Peripheral Devices
- General Review

Section B (SOFTWARE INTERFACE) covers:

- Introduction to "Operating Systems"
- Control Information
- Operator Interface With DOS(Disk Operating System)
- Computer Room Visits (hands-on practicum)
- Stand-Alone Programs
- Compatibility Modes—Emulation
- Course Review

Figure 1A—A no-frills training syllabus (con'td). Note the absence of EAM/unit record equipment training, and a maximum of immediately-applicable job knowledge given in a 54-hour course over a period of 2.5 months (20 sessions). (After CPDA[1 13]). More detailed versions of this and other curricula, syllabi, and lesson plans are available through the ACM Committee on Computing and The Disadvantaged (ACM-CCD).

vious session's work and lessons, taking the instructors' place in essence. The remainder of the class usually jumped in to help the hapless classmate—after waiting an appropriately gruesome few minutes. This 'team' process, a combined 'horizontal' and 'vertical' class structure, and the random 'instructor', all created an involvement within the class. It worked, and beautifully; in fact the class got ahead of the syllabus.

3. The 'fully horizontal 'or 'group/workshop' approach was occasionally attempted by the third pair of instructors. This normally involved the teacher bringing much of the material to the students' attention via lectures and some reading, but required that answers to problems and operations, come from the class as a group. In this particular instance, the structure worked fairly well, the class completed the material on schedule, but as an experiment it was relatively inconclusive. This was partly because the amount of lecture required, and individual help given was more than normally used in a true 'horizontal' workshop situation. That is, in this case, the technique never got a thorough workout.

To summarize: the lecture technique worked fairly well on the brighter students, who expected it as a matter of previous exposure. Their class suffered the greatest number of dropouts, but not from the training technique used.

The 'teams' approach was very effective for both morale and learning. The class was able to cover a few items the others didn't.

The 'interactive' 'fully-horizontal' group organized as a workshop also finished, and reasonably well, certainly comparable to the others in content knowledge. But it is predicted that the stronger extension of that, the new Montessori/workshop group involutional methods should give far better results for disadvantaged people, especially when the staff and the facilities can be structured properly.[15]

The Montessori environment requires careful guidance upon the part of the instructor, and a special quality of allowing the class to explore freely the alternative paths to answers. The instructor, in a sense, must be willing to subdue his usual position of center-focus role, become a part of the discussion, part of the group, almost at their own level. Over a period of weeks, the group should become highly interactive, over the material, over technical and occasionally external life problems, and must be handled carefully. It has been used very effectively for teaching programmers and systems analysts (advantaged),[8] and it is strongly urged that the Montessori techniques and environment be attempted on the disadvantaged in all occupations. At least one project, the Sacramento/ACM Education Committee,[15] is planning to use it for a computer operators' course.

## Training and stability

There is still one more important aspect of training which will aid a project immeasurably: the maintenance of continuity of warm, stable teachers with whom the class can identify, and the assured continuity of class sessions, the same physical facilities, knowing that the class *is* going to meet, and there *will* be a job waiting at the end of the course. Changing classrooms every few weeks and uncertainties of computer time when promised try the motivations of the students (and instructors) sorely, at times. Those situations which had good steady facilities, the same instructors throughout, (usually paid, and professional at teaching itself) have the highest attendance and morale levels. Though these items go almost without saying, the proliferation of volunteer projects impels the author to issue this type of warning, for the sake of everyone involved, particularly the disadvantaged students; they have been through enough instability in life already.

### *Curricula comparison: Methodology of evaluation*

Now that some specific curricula have been presented, we wish to set down some criteria and the method by which we will compare the various content and techniques; to do so we have prepared ourselves in the preceding paragraphs with the job specifications, the required curriculum content, and the training approaches. We consider some of the following points of comparison expanded in Figure 2.

- Is content aimed at the structured, stimulus-response, practical type of course?
- Are the results of the course immediately applicable to a job in a computer installation?
- Does curriculum allow a "flexible tail" so that graduate can go to work in an installation that has computers and operating systems other than the particular one taught?
- Does the course lead into on-the-job training (OJT) easily?

FIGURE 2—Comparison of curricula for computer operator training projects for disadvantaged peoples

| | ACM + Phila. Board of Education | CPDA NYC | Midwest Side NYC | Boston MITRE Corp. | LA/Urban League + IBM + Bank of America | St. Louis Board of Education & IBM | Sanu̇ cisco IBM |
|---|---|---|---|---|---|---|---|
| Practical Structured? | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| Immediately Applicable? | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| Any Computer? | Sort of | No | No | Yes | Yes | Yes | No |
| Theory/Oper. Systems | Yes | No | No | Yes | Yes | Yes | No |
| Programming? | No | No | Separate | No | Some | Yes | Some |
| Basic Language | No | No | No | No | No | Yes | No |
| Higher Language | No | No | COBOL | No | COBOL | RPG, COBOL | COBOL |
| Program Structures | No | No | Yes | No | Yes | Yes | Yes |
| Prog's vs. Op. Systems | Yes | Yes | Yes | Yes | Yes | Yes | Some |
| Flexible Tail—Toward Other Computers | Yes | Little | No | Yes | No | No | No |
| Lead into OJT | Yes | Could | No | Yes | No | No | Could |
| OJT Part of Course? | Yes | No | Some | Yes | No | No | No |
| Training Measurements | | | | | | | |
| —Written Exams | | No | No | Yes | Yes | Yes | Yes |
| —Hands-on Exams | Yes | Some | Yes | Yes | Yes | Yes | Yes |
| —Post-Grad Performance | Not Yet | Call Backs | Inter-views | OJ-by Super-visors | None | None | Check-Back, Verbal |
| "No Failures" Policy? | No | Yes | No | No | No | No | No |
| Support: Remedial | No | Yes | No | Yes | No | No | No |
| Support: Tutoring | Yes | Yes | Some | Yes | No | No | Little |
| Support: Guidance | Some | Yes | No | No | No | No | No |
| No. Students/Teacher | 20 | 16 | 12 or 6 | 4 | 12 or 6 | 11 | 10 |
| Non-English Help? | No | Yes | Yes | No | No | No | No |
| Buddies? Liaisons? | 2 per Student | Yes; Both Counsel-ors and Tutors | No | Some | No | No | No |

Figure 2—Comparison of curricula for computer operator training projects for disadvantaged peoples (Cont'd)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Attrition Rates | | | | | | | |
| Number Started | 20 | 48 | 12 | 4 | 12 | 11 | 21 |
| Number Finished | 19 | 32 | 6 | 3 | 9 | 9 | |
| Number Graduated | 19 | 27 | 6 | 3 | 9 | 9 | |
| Number Placed | 11 | 17 | 4 | 3 | 8 | 9 | |
| Length of Course | | | | | | | |
| Orientation (Hours) | 0- | 8 | 0 | 0 | 0 | 2.5 | 2 |
| Hands-On-Time ( " ) | 25.5 | 10 | | 0 | 40 | 70 | 15 |
| Technical Class ( " ) | 42.5 | 30 | | 0 | 200 | 175 | 30 |
| Other Classes (Remedial) | — | 8* | No | 0 | — | No | |
| Total Class Hours | 68 | 48–56 | | | 240 | 350 | 47 |
| Number of Sessions | 27 | 20 | | | 30 | 40 | 22 |
| Elapsed Time (Weeks) | 12+ | 10 | | 26 | 6 | 8 | 2–4 |
| Hours per Session | (av.) 2.5 | 2 | | 1–5 † | 8 | 5.5 | 2–3 |
| EAM Taught? | Yes | No | No | No | Yes | Yes | Yes |
| Only Key-Punching? | No | Some | – | Yes | No | No | No |
| IBM 360 DOS | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| OS | No | No | No | Yes | No | No | No |
| Other (1401, 7030) | No | No | No | Yes | No | No | No |
| Teacher Stipends | Yes | No | Some | Yes | Yes | Yes | Yes |
| Student Stipends | No | No | No | Yes | No | Yes | No |
| Sponsors | ACM; Phila. Bd. of Education | CPDA Volunteers & City Univ. of N.Y. Guidance Counselors | Also ACM; Then Volunteer + Paid Staff | MITRE Corp. Internally | Urban League + IBM + Bank of America | St. Louis Bd. of Education + IBM | San Francisco-IBM Education Center |
| | | *Optional | | †Class Hours Only | | | |

- How are the results of training measured?
- How is the post-graduate performance measured?
- How much time is given in classroom lecture? In hands-on experience per student?
- Is there a "no failures" policy of teaching?
- How much supportive remedial help is built into curriculum?
- Are non-English students helped?
- How many teachers per student?
- Course content: was EAM/Unit Record equipment taught fully, or was only the keypunch taught (for computer room use)? Was operating system taught as button-pushing course? Or was a 'theory of operating systems' taught in addition, and the relationship between the resident applications programs and the operating system concepts taught?

With these criteria and questions in mind, a chart of the Yes/No/Comments type, (Figure 2) gives a picture to the reader of the various projects, and their relative strengths and weaknesses.

*Performance criteria—Some measures of the projects*

If one were to attempt to measure the results of such training for the disadvantaged, one might look to the annual salaries accruing, of those who obtained jobs, versus the expenditure for the project. MWSDPS[16] suggests that for approximately $14,000 they graduated 28 students of all types (key punch and computer operators, and programmers), and placed 20 of them, for a job value of $98,000 annually, relieving the welfare rolls of eight people at the same time. By the same token, CPDA graduated and placed 17 computer operators, who now earn about $74,000 per year, all for something less than $1,000 cash, but using six teachers, eight guidance counselors, and about a dozen more in placement, tutorial, curriculum development, measurement, and find-a-computer chores, *all volunteer*.[12]

In another performance measure, it is evident that the more deeply into the social problems' fabric a project wishes to penetrate, the more 'underpinning' or training support techniques one must invest energy and staff: these include tutoring, remedial training, high school equivalency aid, teaching English as a second language, and both vocational and life-guidance counseling. The volunteer projects expend 10 to 20 supportive hours for every student class hour.[12] Professional staffs and funded projects tend to use a lower ratio of time (about one-to-one) but this is often balanced by a much larger expenditure of up to approximately $1,000 per student, involving facilities and professionals.

Other measures, such as performance on the job, have been followed up by projects too cursory to warrant reporting at this time.

*The 'shadow programming aide'*

It seems to this writer that there are considerably more jobs available at a slightly higher level of complexity: that of a 'shadow programming aide'.[27] Not just a coder, this person works in shadow relationship to a regular programmer, carrying out some of the more onerous details of programming, such as flowcharting from given coding, setting up debug runs, keypunching, expediting the debug process, or carrying out some of the detailed, but highly supervised coding. This kind of programming technician could very well have real upward mobility, dependent primarily on the trainee's learning rate, general intelligence, interest, and proven ability. The national crush for programmers is far greater currently than for operators; and in addition, the direct personal involvement of a disadvantaged person with a regular, stable, accepting programmer or two, would be one of the best stabilizing entries into the computing field.

## SUMMARY—PITFALLS AND SUGGESTIONS

A number of points may be abstracted from the foregoing, in addition to others not made in the main text:

- The 'typical project' for training disadvantaged people is created by a highly motivated group of computer (and other) professionals, with a desire to get involved, to do something.
- The question each project must face is: just how deeply into the social problems it wants to delve: underemployed, unemployed, educationally-deficited, hard-core unemployed, or whatever.
- Most computer training projects aim at the 19-23 year olds, who are two or three years behind in their education, and won't need very much remedial training to get them through a computer operator (or programming) course and into a meaningful job.

- It is a surprising note, but there seem to be relatively few trainee computer operator courses in the country: mostly these have been given on the job (OJT), or from reference manuals supplied by the manufacturers; leastwise the author has detected very few. This points up a real econ mic advantage for the prospective employers: very little OJT need be spent to start these employees—they're ready to go from Day One. For once, in this case at least, the disadvantaged are probably starting out ahead of their advantaged brethren.

- The first psychological jolt for motivated whites is to discover they will not, most of them, be acting at the actual 'interface' between the disadvantaged and the advantaged community. More of them will have to find psychic reward in support functions, such as finding employers with available jobs, writing curricula, obtaining free computer time, finding teachers and class space, obtaining funds.

- The second psychological jolt comes in discovering how much planning is to be done (or should have been done).

- The third jolt is the marketplace: obtaining job slots requires persuading people outside your little narrow project, and it's tough. Start very early to involve would-be employers, even at the planning stages.

- Courses, so far, have usually involved computer operator training on IBM machines, particularly the 360 series, in DOS or OS (Disk Operating System or Operating System/360). Programming usually means Cobol, rather than basic languages. A way to teach a 'core' curriculum applicable to any operating system for any manufacturer's machine needs to be developed. Such a step has begun to take shape with the ACM-Philadelphia/Board of Education project, using the 'flexible tail' method for phasing to OJT. (Ref. 11).

- The single greatest lack is funds. They are hard to get. Working with the local Board of Education to obtain government funds, or with local industry for underwriting expenses privately, seems to be most effective.

- Volunteer projects usually failed of their total objectives, but succeeded partially; but they tend also to peter out. Plans for real continuity must be built in.

- One cannot stress too much the responsibility we have in changing the lives of would-be trainees.

- The one thing training projects *are* prepared for is attrition; their expectations are that they will graduate 30 or 40 percent of the people who arrive at their door for the first formal class day. It works out that way, though some projects have gotten over 80 percent.

- What is needed for the classes is a new, simple, straightforward text on operating a computer written in the language which the disadvantaged can understand. It may have to be aimed at the ninth grade level, for both the English and the commercial algebra comprehension. It could start with what they know already: numbers systems can come from the numbers game, and you could go on from there. Here's a set of books someone could write, and the whole computing profession may benefit.

- Serious consideration of new, less complex, job descriptions can be attained in the operations and the programming area. It is up to employers and the industry to develop them, in order to include a larger part of our population.

- Finally, the author strongly urges that the computing community initiate a national broad-based organized effort, to develop jobs, regular training projects, and adaptations of training techniques for disadvantaged to enter the field. Two major proposals are now before the ACM Committee on Computing and the Disadvantaged: one from the Sacramento Chapter, one of the few ACM chapters actively pursuing a generalized, funded-plus-volunteers approach; and the other from ACM's Special Interest Group on Computer Personnel Research (SIGCPR) for a Massive Training Project, involving 50 cities and a 1000 students per year, and fully documented, measured research on selection, training, and performance within such projects and their graduates.

All these are part of bringing students from the dark into the light, to help them enter the world of working and earning peoples, to stand on their own two feet with self-respect, and dignity:—in the words of the Prophet Micah, "that they shall sit every man under his vine and under his fig-tree, and none shall make him afraid."

## APPENDIX A

*Computer training projects for the disadvantaged —Brief characterizations and descriptions*

1. NEW YORK CITY: *CPDA* (Computer Professionals Development Association; Fall, 1968. Three parallel classes of 16 men each, Computer Operators only; completely volunteer teachers and staff. First pilot course completed November, 1968, hopefully leading in a funded and/or at least a teachers-paid project. Dr. Allen Morton, IBM/SRI, NYC, President.

- Unique aspects: candidates self-selected based upon own interests after four-session "orientation." Three differing teaching techniques tried: classical lectures, two- and three-man teams, and semi "workshop" group involutional approach, yielding differing results. Strong professional guide ance counseling, heavy tutorial aid, and high school equivalency training available. 48 started, 32 finished, 17 placed. Cost less than $1,000; jobs worth $74,000 per year.

2. NEW YORK CITY: *MWSDPS* (Middle West Side Data Processing School); with Puerto Rican community group; a semi-funded project; started with ACM, then became industry-supported and volunteer. Summer, 1968. L. Barnett, Long Island University, Director.

- Unique aspects: started with 164 off-the-street prospects through advertising, etc. Interviewed applicants for "logic capability", "motivation" and language comprehension. Started 19 programmers in COBOL, graduated 14, placed 9. Started 17 keypunchers, graduated 8, placed 7; started 12 computer operators in IBM 360/DOS, graduated 6, placed 4. Cost about $14,000, and jobs worth $108,000 annually.

3. NEW YORK CITY: *Harlem*; a series of IBM keypunch operator courses organized by W. DeLegall, Columbia University Computing Center. Basic lesson learned: after first course taught basic typing from scratch as preliminary to keypunch training, subsequent classes required candidates to have 40 wpm typing skill before entering.

4. LOS ANGELES: *ULDPTC* (Urban League Data Processing Training Center), 7226 S. Figueroa Ave.; jointly sponsored by League, with professional IBM teachers and IBM donated equipment, in Bank of America donated building. Urban League both selected and placed candidates. This is the most professional and thoroughly equipped computer field training project for disadvantaged in the country. Supported completely by private/industrial funding. John O. Adams, (IBM), Training Director.

- Unique aspects: three parallel courses in keypunching (12 people, four weeks), computer operations (IBM 360/30, DOS, 12 students, six weeks), programming (COBOL, 12 students, 12 weeks). Full daytime staff and students; no student stipends. Has full IBM 360/30 with tapes, discs, printer, card reader/punch, dedicated to project only (i.e., no production, only classes). Runs two years, about 250 students per year. Attrition rate very low (5 percent-20 percent) and placement rate very high.

5. LOS ANGELES: *Maywebb Sciences Corp.* Originally a Watts area volunteer project, spearheaded by Louis Webb. Has graduated programmers primarily. After two years is offering courses on regular paid "private EDP school" basis, and to private industry on government funds.

6. LOS ANGELES: *Operation Bootstrap*: part of Watts area self-help in manufacturing and retail stores. Also started as key punching classes and programming; 47 enrolled in latter, plus remedial training. Founder: Louis Smith.

7. SAN FRANCISCO: *IBM*; computer operator course (EAM + DOS) completely staffed and equipped by IBM's Branch Office Education Center; concentrated on somewhat older group (average age = 28.5 years, top is 45 years), a good portion of whom had seen tab equipment before. High percentage of police records and unemployed. Very successful employment placement. Director, Philip Braverman, IBM San Francisco.

8. SACRAMENTO: *ACM CHAPTER*, Education Committee. Specially planned computer operator training project, to begin classes in September 1969. Includes long, detailed market study of job specifications to determine job placement availability and committed slots for graduates in state government and private industry. Detailed employer/project interaction; year-long careful planning in all phases—systems analysis approach; use of PMS (Program Management System) for scheduling; professional, external structured measures of selection, evaluation of curricula and training performance, being woven in from the start; 'Montessori/workshop' group involutional training techniques; paid teachers; possible

student stipends. Totally ACM-directed project, with industry/government cooperation. Organizer: Elizabeth R. Alexander.

9. BOSTON: *MITRE Corp.*; totally funded by MITRE: the only fully in-house OJT in this survey; see section on Computer operator curricula.

10. ST. LOUIS: *Board of Education and IBM*; 11 students, eight weeks, 5.5 hours per day, in programming and operations; 8 out of 11 graduates are either computer operators or teleprocessing clerks; 1968 Summer project, not being repeated. Funded by Board of Education (Dr. Lawson, Treasurer) and staffed by regular IBM Systems Engineers, as professional teachers. Students received stipends, and were chosen from group of 60 *not* planning to go on to college and with average grades. Former Director: Ron Dobies, IBM/DPD, Clayton, Missouri.

11. PHILADELPHIA: *ACM and Board of Education*; Delaware Valley Chapter ACM in cooperation with Thomas Edison High School counsellors; an after-school hours project (two days/week) for 20 volunteer students; paid teachers, two "buddies" per student volunteer from ACM; runs about 25 weeks; includes EAM through full 360/DOS operations; furnishes some OJT at end of course to dovetail with prospective employers' non-IBM computers. Director, Milton Bauman, ACM and Price, Waterhouse and Co., Phila.

## REFERENCES

12 J P GILBERT D B MAYER
*Experiences in selection of disadvantaged people into a self computer data processing training program*
Proc Seventh Annual Conf on Computer Personnel Research ACM-SIGCPR (to be published) 1969

13 J BURROWS
*Report on Mitre OJT training project*
Personnel communication to the author

14 W A DE LEGALL
*Teaching techniques and quality education/training for the disadvantaged*
Proc Seventh Annual Conf on Computer Personnel Research ACM 1969 to be published

15 E R ALEXANDER
*Montessori techniques applied to programmer training in a workshop environment*
Proc SJCC Vol 34 1969 373-379

16 *Data processing training for the underemployed: an evaluation of an experiment*
The Diebold Group Inc 1969 27

17 U S Dept of Labor Bureau of Labor Statistics Nov 1967 on a Census of the Negro Community

18 J EYELER
*From welfare rolls to over $7,000 a year in four months*
Datamation Mag April 1969 175-177

19 M BAUMAN
*Computers and the underprivileged*
Proc SJCC Vol 34 1969 35

20 J J DONOVAN
*A program for the underprivileged and the overprivileged in the Boston community*
Proc SJCC Vol 34 1969 36

21 W B LEWIS
*What the JOBS program is all about*
Proc SJCC Vol 34 1969 37

22 A L MORTON JR
*Computers and the underprivileged*
Proc SJCC Vol 34 1969 38

23 J SEILER
*Experimental and demonstration manpower projects*
Proc SJCC Vol 34 1969 38

24 H GRIFFIN P GRAVELLE
*Report on the philosophy and mechanics of the urban education committee of Philadelphia*
Proc Seventh Annual Conf on Computer Personnel Research Assoc for Computing Machinery 1969 to be published

25 *California State Personnel Board: Specifications for computer operator trainee and for computer operator*
1969

26 *The Mitre Corp: Example of an in-house computer operator training using OJT techniques*
File Memorandum Outline ACM Committee on Computing and the Didsavantaged May 1969

27 D B MAYER
*A suggested new entry-level job for the disadvantaged: Shadow programming aide*
File Memorandum Job Spec ACM-CCD 1969

# The Q approach to problem solving

*by* J. D. McCULLY

*TRW Systems*
Redondo Beach, California

## INTRODUCTION

The problem of determining derivatives on a digital computer has received a great deal of attention for several years. Some exotic systems have been developed and numerous papers have treated the problem. In 1964 it was suggested by Wengert[1] that the chain rule could be applied to values for the determination of derivatives.

This general concept has served as the basis for a series of programs developed at TRW Systems. It has been expanded to permit the essentially simultaneous computation of first and second partial derivatives with respect to several independent variables. Second partials are especially valuable in optimization problems, and excellent results have been obtained with this technique. The first program written at TRW some years ago to apply Wengert's chain rule concept was called ROP (for Restricted Optimization Program) and has been used to optimize sets of algebraic equations. After some experience with this program it was decided that a complete system should be devised to permit wider application of the technique to problems where partial derivatives would be of value. The system was initially named CUE, for Computer Utility for Engineers, but was recently renamed Q in deference to another system named CUE.

The intent was to make Q essentially a computer operating system. On the other hand, it was to be used within an already existing operating system (SCOPE 2.1) on TRW's CDC 6500 machine without modification to the existing system. A good discussion of this type of system is found in Glass.[2] The consequence was necessarily some added overhead operating cost, but it was hoped that two factors would offset this added cost. One of these factors was the planned ma chine-independent characteristic of the Q system which essentially uses only FORTRAN and FORTRAN routines (including I-O). In practice, some of the machine-oriented functions of the SCOPE operating system proved impossible to resist and conversion to another machine may be less easy than was originally planned.

The second factor that would make Q attractive despite the increased machine time was the inclusion of several unique features in the system. The most important of these features is the above mentioned partial derivatives. Another is dynamic storage, and a third feature of interest is a macro processor for the input language. With this feature the system is suitable for use by the engineer who is more or less familiar with FORTRAN and wants his job done quickly even at the expense of some extra machine time.

### *Sample problems*

Before the structure and characteristics of the Q system are described in detail, it may be useful to give some examples of the kind of problem for which it has proved most useful. These examples are taken from INTRODUCTION to SLANG.[3] In general it can be said that Q is suitable for mathematically complex problems. It has been designed to relieve the user of most of the complex calculations involved and to provide him with a short turnaround time that makes practical a series of alternate approaches or formulations.

As an essential part of making Q user-oriented, a high-level language called SLANG has been evolved to allow easy communication with the computer by engineers with little programming knowledge. For purposes of the sample problems it is necessary to keep in mind that the problem statements shown are written in SLANG. The convenience of formulating problems in this way will be apparent.

The first example illustrates the use of SLANG for solving a typical optimization problem with nonlinear implicit equations imbedded in the engineering model. The problem is to minimize the weight of a three-stage liquid rocket vehicle boosting a payload from the surface of Mars. The optimum values of thrust level and burn time for each stage are to be determined for the specified mission. Total burn time, total velocity increment, and payload weight are given. The SLANG statements required to solve this problem are shown in Figure 1.

In this problem, the quantity being minimized is WTØT the statement

ØPTIMIZE WTØT (1)

identifies the payoff function and establishes an optimization loop which ends with the second END LØØP card. The statement

INDEPENDENT THRUST(2), THRUST (3), TBURN(1), TBURN(2) (2)

designates thrust levels of two stages and burn times of two stages as independent variables which are being determined by the optimization. Equations G1 and G2 are being solved to constrain the solution such that total velocity increment and burn time match specified values. The statement

SOLVE G1, G2 (3)

identifies the implicit simultaneous equations being solved and establishes an equation solving loop which ends with the first END LØØP card. The independent variables of the SØLVE loop are identified by the statement.

INDEPENDENT THRUST(1), TBURN(3) (4)

Even though they are expressed in terms of intermediate variables, the equations G1 and G2 are equivalent to the ultimate form

$$G1 = G1\,(\mathrm{THRUST}\ (1), \mathrm{TBURN}\ (3)) \qquad (5)$$

$$G2 = G2\,(\mathrm{THRUST}\ (1), \mathrm{TBURN}\ (3))$$

```
    VARIABLE ISP(3),ISPVAC(3),TBURN(3),THRUST(3),XIP(3),
*            WPRØP(3),WSTAGE(3),STRFAC(3),DELV(3),MR(3)
  1 READ DATA
    ØPTIMIZE WTØT
       INDEPENDENT THRUST(2),THRUST(3),TBURN(1),TBURN(2)
       ØLIMITS(FPRIN = 0)
       SØLVE G1,G2
          INDEPENDENT THRUST(1),TBURN(3)
          DLVTØT = 0
          W = WPAYLD
          TBTØT = 0
          DØ FØR L = 1 TØ 3
          I = 4-L
             ISP(I) = ISPVAC(I) * (1 - XIP(I))
             WPRØP(I) = THRUST(I) * TBURN(I) / ISP(I)
             WSTAGE(I) = 0.0234 * THRUST(I) + WPRØP(I)
*            + 1.255 * WPRØP(I) ** 0.704 + 4
             STRFAC(I) = WPRØP(I) / WSTAGE(I)
             W = W + WSTAGE(I)
             MR(I) = W / (W - WPRØP(I))
             DELV(I) = GC * ISP(I) * LØGN(MR(I))
             DLVTØT = DLVTØT + DELV(I)
             TBTØT = TBTØT + TBURN(I)
       REPEAT
          G1 = DLVTØT - DELVIP
          G2 = TBTØT - TBTIP
       END LØØP
       WTØT = W
       PRINT VARIABLES
    END LØØP
    GØ TØ 1
    END
     DATA
     THRUST=5400,1237,317,TBURN=142,127,131,GC=32,174,WPAYLD=
     DELVIP=2.8E4,TBTIP=400,ISPVAC=315,315,315,XIP=0,0,5E-3,
     $END
```

Figure 1—SLANG formulation of sample optimization problem

The purpose of the SØLVE loop is to find the values of THRUST (1) and TBURN (3) that satisfy $G1 = 0$ and $G2 = 0$. Engine performance and vehicle weight quantities are computed in a loop beginning with the statement.

DØ FØR L = 1 TØ 3 (6)

and ending with

REPEAT (7)

The equations between these two statements are used three times, one time for each of the three stages. Two characteristics of SLANG should be evident from this example. One is that the SLANG expressions used to describe the engineering model closely resemble those of FØRTRAN. The other is that numerical algorithms for optimization and nonlinear equation solving are invoked using the commands ØPTIMIZE and SØLVE.

The total running time for this problem was eight seconds on the CDC 6500. The printout of the solution is shown in Figure 2.

The second example demonstrates how a SØLVE loop can be used to match an integration boundary

```
                                    Variable Values
DELVIP  2.80000E+04  DELV   1.06014E+04          9.30107E+03          8.09754E+03  DLVTOT  2.80000E+04
GC      3.21740E+01  G1     0.           G2      0.           ISPVAC  3.15000E+02          3.15000E+02
        3.15000E+02  ISP    3.15000E+02          3.15000E+02          3.13425E+02  I       1.00000E+00
L       3.00000E+00  MR     2.84635E+00          2.50361E+00          2.23222E+00  STRFAC  8.51072E-01
        7.95069E-01         7.14542E-01  TBTIP   4.00000E+02  TBTOT   4.00000E+02  TBURN   1.51068E+02
        1.33079E+02         1.15853E+02  THRUST  5.11094E+03          1.27746E+03          3.28288E+02
WPAYLD  5.00000E+01  WPROP  2.45111E+03          5.39694E+02          1.21347E+02  WSTAGE  2.88002E+03
        6.78801E+02         1.69824E+02  WTOT    3.77865E+03  W       3.77865E+03  XIP     0.
        0.                  5.00000E-03
```

Figure 2—SLANG printout of results from problem shown in Figure 1

```
/SØLID RØCKET ENGINE START-UP TRANSIENT PRØBLEM
/  THE PURPØSE ØF THIS PRØBLEM IS TØ DETERMINE
/  THE PERCENTAGE ØF EQUILIBRIUM CHAMBER PRESSURE
/  ATTAINED BY AN END BURNING SØLID RØCKET ENGINE
/  AT A SPECIFIED TIME (TSPEC) DURING ITS STARTUP
/  TRANSIENT
/  THE PRØBLEM INVØLVES INTEGRATIØN, BØUNDARY CØNDITIØN
/  MATCHING, AND HAS A SØLVE LØØP
READ DATA
PCEQ = (12/32.174 * RHØP * CSTARO * A * K) ** (1/(1 - N - Q))
     SØLVE CØNST
     INDEPENDENT PCSPEC
              FAC = VC / (GAM ** 2 * AT * 12)
              LET TINTEG = INTEGRAL (1 / (CSTARO * PC **
*                  Q * PC * (RHØP * CSTARO * PC ** Q * A *
*                  K * PC ** (N -1) * 12 / 32.174 - 1)),
*                PC = PCIG TØ PCSPEC IN 10 STEPS)
              TCØMP = FAC * TINTEG
              CØNST = TCØMP - TSPEC
              PRINT VARIABLES
     END LØØP
PERCNT = PCSPEC * 100 / PCEQ
PRINT VARIABLES PERCNT
STØP
END
 DATA
 TSPEC = 0.5,
 PCSPEC = 1500,
 PCIG = 700,
 RHØP = 0.064
 CSTARO = 3320,
 A = 4.4 E-4,
 K = 172.65,
 N = 0.745,
 Q = 0.015,
 VC = 220,
 GAM = 0.66175,
 AT = 0.35,
 $END
```

Figure 3—SLANG formulation of boundary matching problem

condition. The complete set of input is shown in Figure 3.

The expression in the argument of the integration statement is an equation for $dt/dP_c$ (where $t$ = time, $P_c$ = chamber pressure) during the start up transient of a solid rocket engine. The problem is to determine the value of chamber pressure at a specified time. This value is the upper limit of integration, and is being computed such that the integrated time (TCØMP) matches the specified time (TSPEC). That is, when the value of the constraint, CØNST, is zero, the upper integration limit PCSPEC is the value of chamber pressure at TSPEC. The final calculation of PERCNT computes the percentage of equilibrium chamber pressure, PCEQ, achieved at time TSPEC. PCEQ is computed from input data. The lower limit of integration, PCIG, is the ignition pressure, and is an input constant.

## *Strucutre of the Q system*

The Q system is basically a Complier/Interpreter type package with the four major elements of the system shown in Figure 4. The user's input language (SLANG) is converted by a set of system-supplied macros into the MODTRAN language. The MODTRAN compiler then converts this language into an assortment of pseudo instructions and some associated tables. These are processed by the link editor before going to the interpreter for execution.

With this system it is possible to omit the macro processor if the user chooses to write directly in MODTRAN. On the other hand, a user might wish to use only the macro processor to perform some transformations on BCD data.

The ML/I processor was originally designed by P. J. Brown[4] of Cambridge University, who supplied the logic to TRW. The processor was converted to FORTRAN with little difficulty, and this version was included in the CUE system for making an initial pass at the input of non-programmer users. It was found that the average engineer in a hurry (for whom the system was designed) was unwilling to take the trouble of writing his own macros. Ideas for suitable macros were solicited from potential engineer users, and the resulting language was christened SLANG. Additions are continuously being made to SLANG to make it more useful. At one time it was planned to have four

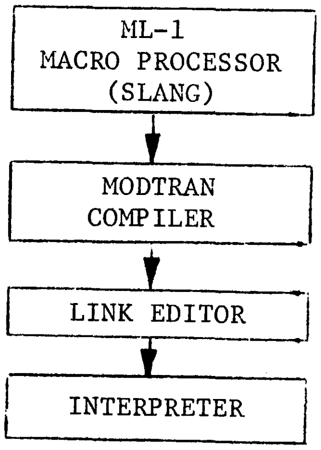


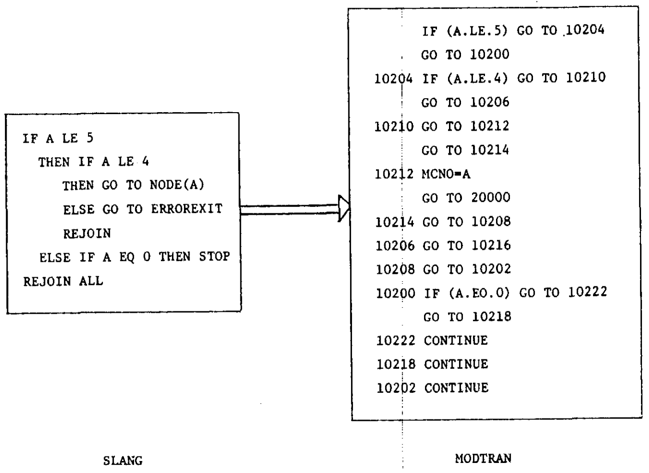
Figure 4—Basic Q system elements

"dialects" of SLANG of increasing degrees of sophistication, but this idea was abandoned in favor of a single version.

An example of how the processor converts SLANG macros to MODTRAN is shown in Figure 5. It is worth noting that the writing and debugging of macro definitions is considerably easier than would be the modification of the MODTRAN compiler itself. The programmer need in general be concerned only with the particular macro definition he is working on, and both his inputs and his outputs are in BCD.

It was originally planned to incorporate some of the more popular SLANG variations into MODTRAN, thus reducing processing time; unfortunately this project has been continuously postponed because of more pressing work. The more recent versions of the Q system allow for relocatable subroutines, which have served to reduce machine time considerably. Previously an illusion of subroutines was created by suitable macros, but it was necessary to process the user's entire input deck each time the equations were modified.

The MODTRAN language bears a strong resemblance to FORTRAN or BASIC, since it was designed by FORTRAN programmers. Algebraic statements are essentially the same, and DO loops are provided that have the same function except that they provide for backward stepping when desired. Arrays are as in FORTRAN except that they are limited to two indexes. READ and WRITE statements are similar, as are FORMAT statements. All variables are floating point as in BASIC, and corrections are automatically made for round-off errors on comparisons.



```
IF A LE 5
  THEN IF A LE 4
     THEN GO TO NODE(A)
     ELSE GO TO ERROREXIT
     REJOIN
  ELSE IF A EQ 0 THEN STOP
REJOIN ALL
```

```
      IF (A.LE.5) GO TO 10204
      GO TO 10200
10204 IF (A.LE.4) GO TO 10210
      GO TO 10206
10210 GO TO 10212
      GO TO 10214
10212 MCNO=A
      GO TO 20000
10214 GO TO 10208
10206 GO TO 10216
10208 GO TO 10202
10200 IF (A.EO.0) GO TO 10222
      GO TO 10218
10222 CONTINUE
10218 CONTINUE
10202 CONTINUE
```

Figure 5—Example of SLANG/MODTRAN conversion

Some MODTRAN statements are unusual, as for example EXECUTE label, which will cause a transfer to the label. When a JUMPBACK statement is encountered, control is tranferred to the statement following the EXECUTE label.

The FORTRAN subroutine concept is used in MODTRAN, but the COMMON method of communicating between subroutines was eliminated in favor of using the names of the variables themselves to communicate locations, as in BASIC and other languages. Another provision is that a variable can be typed as LOCAL to a particular subroutine, permitting subroutines to be written independently. The FORTRAN concept of calling sequence/argument list is used for communication between such subroutines, so that MODTRAN subroutines may be written and placed in the system library for general use.

The MODTRAN compiler has no provision for user-written functions (arithemetic or other), which makes it possible to determine an indexed variable even though no suitable allocation statement has appeared. When the compiler encounters what appears to be an array (which could be a misspelled system function), it processes the indices and assumes that by the time the statement is executed another statement making the allocation for the array will have been previously executed. The allocation statement can be either GLOBAL or LOCAL. For example, the statement:

GLØBAL X (NRØW, NCØL), Y (10), Z (8)

will cause the release of any arrays previously associated with X and Y and the allocation of ten words to Y as well as the generation of an array NRØW rows by NCØL columns for X. Such statements are executable, and once executed will apply to all other subroutines where the variables X and Y appear as globals. The variable Z in this statement is only given a global assignment by the compiler and that portion of the statement is not executable. If the compiler encounters a variable not defined as GLØBAL or LØCAL it assigns the variable to the nominal category previously defined by the user (normally GLØBAL).

### *Generation of partial derivatives*

Perhaps the most interesting feature of the Q system is the way in which partial derivatives are treated. The MODTRAN language provides for specification of three levels of partials:

$$\begin{aligned}&\text{NO PARTIALS}\\&\text{FIRST PARTIALS List}\\&\text{SECOND PARTIALS List}\end{aligned} \tag{9}$$

In these statements, List specifies which variables are to be the independent variables. An INDEPENDENT List statement might also be used for this purpose. A typical set of statements might be:

$$\begin{aligned}&\text{SECOND PARTIALS X, Y, Z}\\&F = Y * X/Z\\&D = F * F\end{aligned} \tag{10}$$

These statements will cause the dependent variables D and F to be evaluated and all of the first and second partial derivatives of these two variables with respect to X, Y, and Z will be computed. The resulting storage requirements can become quite large; in the case of three independent variables one word is required for the value, three for first partials, and six for second partials, making a total of ten words (see equation 11). In the case of 15 independent variables 136 words of storage are required for each dependent variable. The system tries to hold down the total storage required by returning the partial storage to the free area wherever possible. We are considering a scheme to reduce the number of words required in the case of a dependent variable that is not a function of all the independent variables.

The actual operation of computing partial derivatives is carried out by the interpreter in the course of evaluating the given expressions of the problem. This evaluation consists essentially of a sequence of operations, which may be unary (performed on a single variable), for example SIN (X) or binary (performed on two variables), for example X*Y. The result of an operation either becomes one of the variables going into the next operation or, if the sequence is complete, the result is stored as the answer in the appropriate location. An operation is performed by the interpreter causing a transfer to one of the appropriate subroutines. Each subroutine has either one primary input (unary), or two primary inputs (binary), and a single output. The inputs (operands) may or may not have partials, and if they do it may be necessary to compute only first partials or both first and second partials. Consider the division operator, for example; either or both the divisor and dividend may or may not have partials, leading to four different possible cases. Each case is different with respect to how the partials of the resultant variables are computed, and four separate subroutines have been written for the division operator; the appropriate subroutine is selected by the interpreter during the execution of the user's program. If an equation is evaluated several times, it is entirely possible that a variable may have partials during one evaluation and none during another, in which case the appropriate subroutine would be executed during each evaluation. At the time that the link edit is performed every variable is given a core location assignment. If the variable has no partials then the value associated with the variable is stored in this location. If, however, during the execution of the model the variable develops partial derivatives by being a function of variables which have partials, then a vector is opened for the variable and the initial location replaced by a pointer to this vector. As an illustration, consider the following sample vector for a variable F when there are three independent variables X, Y, and Z:

$$\begin{aligned}F, \frac{\partial F}{\partial X}, \frac{\partial F}{\partial Y}, \frac{\partial F}{\partial Z}, \frac{\partial^2 F}{\partial X \partial X}, \frac{\partial^2 F}{\partial X \partial Y}, \frac{\partial^2 F}{\partial X \partial Z},\\ \frac{\partial^2 F}{\partial Y \partial Y}, \frac{\partial^2 F}{\partial Y \partial Z}, \frac{\partial^2 F}{\partial Z \partial Z}\end{aligned} \tag{11}$$

All of the variables which have partials will have similar associated vectors. The independent variables will each have such a vector where all of the partials are zero except for the one corresponding to the derivative of the independent with respect to itself where a value of one will be stored. When an INDEPENDENT statement is encountered all of the vectors which happen to be active at that point are deleted and a new set of independent vectors set up. As the run progresses new dependent vectors will be allocated.

In MODTRAN statements for unary operations, the subroutines tend to be similar except for the three lines for the evaluation of F, S1, and S2 (see below for definition of S1 and S2). In the example of Figure 6, SINX is used as the name of the interpreter subroutine for evaluating the sine of a variable. NUMIND indicates the number of independent variables, E is the operand vector, and F is the resultant vector.

There would of course be similar routines for COS, EXP, TAN, etc., which might appear in the user's input. In the general case all of these subroutines would be identical except for F, S1 and S2. Suppose oper corresponds to the unary operator that is being used, then F, S1 and S2 can be expressed in general as follows:

```
        SUBROUTINE SINX(E,F)
        DIMENSION
        CØMMØN/NUMIND/NUMIND
1       F(1) = SIN(E(1))
2       S1= COS(E(1))
3       S2= -SIN(E(1))
        M=NUMIND
        DO 20 K=1,NUMIND
        IF (FIRST) GØ TO 20
        S3=S2*E(K)
        DO 10 L=1,K
        M=M+1
10      F(M)=E(M)*S1+S3*E(L)
20      F(K)=F(K)*S1
        RETURN
        END
```

Figure 6—Sample interpreter subroutine for unary operation

$$F = \text{oper}(E)$$
$$S1 = \frac{\partial \text{oper}(E)}{\partial E} \qquad (12)$$
$$S2 = \frac{\partial^2 \text{oper}(E)}{\partial E^2}$$

Should it be necessary to evaluate only first partials then at the time each of the subroutines is executed the logical variable FIRST will be set to true and the computing of the second partials will be bypassed.

Binary functions vary considerably, but an example of this type of function is given in Figure 7 for the multiplication operation. D and E are the operands and F is the resultant vector.

Perhaps it would be useful to demonstrate the manner in which the equations of the MUL routine were derived. Assuming for purposes of explanation that X & Y are the only independent variables then we know that

$$F = D \cdot E \qquad (13)$$

```
        SUBROUTINE MUL(D,E,F)
        DIMENSION D(1),E(1),F(1)
        CØMMØN/NUMIND/NUMIND
        M=NUMIND
        DO 20 K=1,NUMIND
        DO 10 L=1,K
        IF(FIRST) GØ TO 20
        M=M+1
10      F(M)=D(M)*E(1)+E(M)*D(1)+D(K)*E(L)+D(L)*E(K)
20      F(K)=D(1)*E(K)+D(K)*E(1)
        F(1)=D(1)*E(1)
        RETURN
        END
```

Figure 7—Sample interpreter subroutine for binary operation

Then from any table of derivatives

$$\frac{\partial F}{\partial X} = D.\frac{\partial E}{\partial X} + \frac{\partial D}{\partial X} \cdot E \qquad (14)$$

while

$$\frac{\partial F^2}{\partial X \partial Y} = D.\frac{\partial^2 E}{\partial X \partial Y} + \frac{\partial D.}{\partial Y}\frac{\partial E}{\partial X} + \frac{\partial D.}{\partial X}\frac{\partial E}{\partial Y} + \frac{\partial^2 D.}{\partial X \partial Y}E \qquad (15)$$

The reader should be able to convince himself that the statement at label 20 on Figure 7 corresponds to (14) while the statement at label 10 corresponds to (15). It should also be possible to place these statements in the context of a generalized number of independent variables by referencing equation No. 11.

Tabular function defined by arrays of input data are handled by a system routine which fits a polynomial to the data and then assumes that the derivatives of the polynomial correspond to those of the function. This is of course rather cumbersome and the results may not be accurate for many functions.

### *System supplied routines*

In addition to the usual system-supplied routines such as those illustrated above, the Q system attempts to provide rather elaborate sets of routines which are called algorithms. These routines should remove some of the burden off the user to provide a method of solution. They are kept in the Q FORTRAN library and are called as needed. Since one of the main features of the system is the ability to take partial derivatives, it is not surprising that most of these routines are built around this capability. The most important and most frequently used of these algorithms are called SOLVE, OPTIM, and INTEG.

The SOLVE algorithm makes use of the Newton-Raphson technique in order to drive specified functions to zero. In order to do this it is necessary to evaluate the first partial derivative of the functions and apply correction factors to the independent variables based on this information until the convergence criteria is reached. Since it is possible to obtain first partial derivatives by numerical techniques this method of solving functions is rather common. The partials of the Q system should be more accurate, however, especially in the neighborhood of singularities.

The optimization function is initiated by writing MAXIMIZE, MINIMIZE, or CRITICALIZE followed by the variable to be optimized and an INDEPENDENT statement for the variables the system will vary in an attempt to find a solution. The partial derivatives play a major role in this algorithm. Originally the system made use of Lagrangian multipliers in conjunction with the Newton-Raphson technique for optimization, but this method has been superseded by a modified version of rotational discrimination, as described by Law and Fariss.[5]

The INTEG algorithm is used to integrate a set of simultaneous differential equations by a fourth-order Runge-Kutta method. It can be combined with the SOLVE algorithm to solve two-point boundary value problems, as in the second SLANG example given earlier. In this case the INTEG routine is imbedded within a SOLVE loop, where the solution to the SOLVE operation is the end points to match certain expressions. Other routines are available to save and restore partial derivatives, to add and delete independent variables, to input or printout all global variables, etc.

*Implementation of the system*

As implemented on the CDC SCOPE system Q requires two back-to-back executions under SCOPE with a compilation by the SCOPE FORTRAN compiler separating the executions. The user need not be aware of these efforts in his behalf, however, as he submits one job and gets one output. It is even possible to place the SCOPE control cards necessary to run the Q system onto a file, along with the various other files required by the system, so that the user need only see a few of the SCOPE control cards.

In the first execution under the Q system a basic monitor surveys the user control cards to determine the objective of the decks which the user supplies. Thus in one run the user might have some SLANG decks to be sent via the ML/I processor to the MODTRAN compiler, some MODTRAN decks which would go directly to that compiler, some FORTRAN decks for compilation by the SCOPE FORTRAN compiler when it is called in between the executions, and perhaps even some FORTRAN and/or MODTRAN relocatable decks. Control cards are intermixed with other input and some action is normally taken immediately with the cards following a control card until the next control card in encountered. Sometimes a set of cards is sent directly to a processor, such as a MODTRAN deck going to the MODTRAN compiler, while in other cases it is necessary to place the deck on a file for later processing, such as a FORTRAN deck. The flow of operations is shown in Figure 8.

Once all the user's input except for data cards has been read and either processed or assigned, the link editor is called in to tie together the various MODTRAN routines. The link editor assigns all of the variables to their final locations in the data portion of the bucket and performs the required relocation of the pseudo instructions. An attempt is made to satisfy all of the externals referenced from MODTRAN routines with MODTRAN entry points, including a search of the Q MODTRAN library file. The references which are still unsatisfied are assumed to be for FORTRAN routines and a search of the Q FORTRAN library file is performed. Any routines found there are pulled off for loading by the next execution. At this point the link editor writes a FORTRAN routine which will be compiled by the CDC FORTRAN compiler. This routine consists of a computed GO TO followed by a call to each of the routines which it has determined are FORTRAN. For example it might write:

```
  SUBROUTINE CALLI
  COMMON/N/N
  GO TO (1,2),N
1 CALL INTEG                    (15)
  RETURN
2 CALL BROP
  RETURN
  END
```

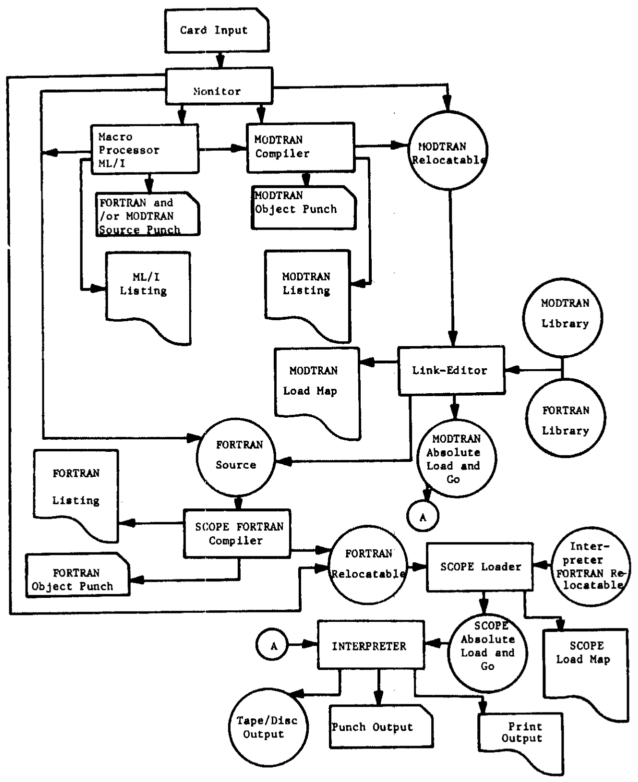


Figure 8—Flow diagram of the Q system

Actually the routine CALLI will be more complicated than this example, since the user is allowed to have arguments to these FORTRAN routines. The basic concept is, however, that this is the manner in which it is made posible to call a FORTRAN routine from a MODTRAN routine. Should the user, for example write

CALL INTEG (16)

in MODTRAN he will in actual fact be calling subroutine CALLI with N set equal to 1. Since the routine CALLI is placed in the input stream to the FORTRAN compiler the user receives a listing of this routine in the middle of his output. It was not deemed worthwhile to try to suppress this listing, since the user might very well be compiling some of his own routines on the same call to the FORTRAN compiler.

After the link editor has relinquished control to the FORTRAN compiler and that processor has completed its task, the second execution of the user's job begins. This consists of a loading of the Q interpreter and all of the FORTRAN routines which have been collected by the link editor on the previous execution and placed on the FORTRAN relocatable file. The nature of this core load varies radically depending on what the user requires. Control is initially passed to the main MODTRAN routine but after that the user is on his own.

During execution of a MODTRAN routine, the pseudo instructions put out by the MODTRAN compiler are being interpreted. As is usual with interpretive schemes, quite a bit of control can be exercised in making sure that the user is not getting into trouble and in taking some appropriate action when he attempts to do something which would be improper.

There are three user data areas in the Q system: variables, arrays, and partials. The three areas are rather heavily intertwined with pointers, a pointer being distinguishable from a value by the fact that it is a positive integer while a value is a normalized floating point number. Initially only the variable area is assigned (by the link editor) and the interpretation of the user's program causes the buildup of the other two areas. Thus suppose the user says

GLOBAL X(10) (17)

where X was previously only a value. An array will be opened in the array area and the location at which X was assigned will be replaced by a pointer to the array. A double tag system as described by Knuth[6] is used for the allocation of arrays, a system which allows a good method of returning variable length arrays to the free area. Two more words are used to specify the dimensions on the array, causing the use of four words in addition to the actual size of the user's array. When an array is released, the two words which were used for indexing are replaced by linking pointers to facilitate the search for free areas of adequate size. The user of course need not be aware of this process when he opens or closes an array.

It is also frequently the case that a variable will not only have a value associated with it but will have some partial derivatives. In this case the location of the variable, or the indexed location within its array, is replaced by a pointer into the partial area. At the location in the partial area the value and the associated partials are stored. Some rather complicated chaining-down pointers may result before the desired location is finally achieved; but normally if the user is taking partials he will be spending most of his time during execution doing just that, computing partials, and the time spent on pointers will be relatively small. It was also necessary to make some provision for returning these partial vectors to the free area, but this is a rather simple matter since all of these vectors are of the same length.

Additional complications are entered into the system when the user performs such operations as saving partials and beginning a new set. This is basically performed by closing off the current partial area and opening up a new one. A swapping of pointers with values occurs so that the partials can be restored later.

## SUMMARY

No claims are made that the Q system is a direct challenge to other computer systems. It does, however, offer anapproach to some rather difficult problems. As was pointed out earlier, it is easy to introduce modifications into the SLANG language, a characteristic which is not common to programming languages. It is also rather easy to introduce new algorithms into the system, thereby expanding its problem solving capability. It is hoped that the Q system constitutes a basis for further development along these lines since the user is frequently denied this flexibility in a computer system.

## ACKNOWLEDGMENTS

Bob Kennedy helped in the preparation of this article and Dave Adamson provided the sample SLANG problems along with the discussion of them.

## REFERENCES

1 R E WENGERT
*A simple automatic derivative evaluation program*
C A C M Vol 7 1964 463-464

2 R L GLASS
*An elementary discussion of compiler/interpreter writing*
Computing Surveys 1 1969 55-77

3 D ADAMSON
*Introduction to SLANG*
TRW Doc 99900-6672-R0-00 1968

4 P J BROWN
*Macro processors and their use in implementing software*
Thesis Univ Math Lab Cambridge England 1968

5 V J LAW R H FARISS
*Rotation discrimination for optimization with limits on the variables*
Preprint 19B Second Joint AICHE-IIQPR Meeting
May 19-22 1968 Tampa Fla

6 D E KNUTH
*The art of computer programming*
Addison Wesley Publishing Co Vol 1 Chap 2 1968 p.442

# Self-contained exponentiation*

*by* NANCY W. CLARK and W. J. CODY

*Argonne National Laboratory*
Argonne, Illinois

## INTRODUCTION

The traditional implementation for floating-point exponentiation, $x$ raised to the $y$ power, is to compute $exp\ (y\ \ell n(x))$ using standard subroutines for the logarithm and the exponential function. While it is possible to provide extremely accurate subroutines for these latter functions, we shall shortly see that this is seldom done. Even in those rare cases where excellent subroutines are available, the exponentiation routine, for sound theoretical reasons, is poor. In this paper, we present brief statistics indicative of the quality of these three subroutines in the basic Fortran libraries provided by various manufacturers, a detailed error analysis for exponentiation, and a method for exponentiation via self-contained subroutines.

In the following discussion we will use the term *exponentiation* to refer to $x^y$ where we will always assume $x > 0$. The term *exponential* will refer to $c^y$ where $c$ is a fixed constant base, usually either 2 or $e$.

### *The present situation*

With the cooperation of a number of different individuals and computing centers, we ran some simple tests on the exponential, logarithm and exponentiation subroutines in the basic Fortran libraries on eight different computers representing six different manufacturers. The only version of the single-precision library on the CDC-3600 available to us contained routines we had written according to the methods to be described and does not necessarily represent the manufacturer's library. We also tested our own version of the library for the IBM S/360 in addition to the standard library.

These tests were not intended to be complete certifications of the routines tested, but were designed to lightly probe areas where such subroutines are most likely to have trouble. The tests consisted of computations with a series of arguments exactly representable in binary notation. The corresponding function values were output in octal or hexadecimal form and compared against similar computations in 96-bit arithmetic on a CDC 6400. The computations involved were:

$$\exp(n) \qquad n = 40(1)88,$$

$$\ell n(x) \qquad x = .25(.015625)2.0,$$

$$x{*}{*}y \qquad (x,y) = \begin{cases} \left.\begin{array}{l}(2^n,\ 22 - n)\\ (4^n,\ 11 - n/2)\end{array}\right\}\ n = 0(1)22,\\ \left.\begin{array}{l}(2^n,\ 44 - 4n)\\ (.75 \times 2^n,\ 46 - 4n)\end{array}\right\}\ n = 1(1)11.\end{cases}$$

The test results are summarized in Table I.

Certain of the computers used have either octal or hexadecimal floating-point arithmetic. On these computers, a mantissa can be properly normalized and still have the first two or three bits zero. This accounts for the apparent tabular discrepancies between the sum of the maximum number of bits in error and the minimum number of correct bits, and the total number of bits in the mantissa on these machines.

* Work performed under the auspices of the U. S. Atomic Energy Commission.

TABLE I—Accuracy Test Results

| Machine and Subroutine | Single-Precision | | Double-Precision | | Machine and Subroutine | Single-Precision | | Double-Precision | |
|---|---|---|---|---|---|---|---|---|---|
| | M | N | M | N | | M | N | M | N |
| Burroughs B-5000 | (39 bit mantissa) | | (78 bit mantissa) | | IBM 360/75 IBM library | (24 bit mantissa) | | (56 bit mantissa) | |
| EXP | 9 | 30 | 8 | 69 | EXP | 1 | 21 | 7 | 49 |
| LN | 3 | 35 | 7 | 71 | LN | 3 | 20 | 3 | 52 |
| X**Y | 7 | 31 | 11 | 67 | X**Y | 10 | 14 | 10 | 46 |
| Control Data 3600 | (36 bit mantissa, Argonne library) | | (84 bit mantissa, CDC library) | | IBM 360/75 Argonne library | (24 bit mantissa) | | (56 bit mantissa) | |
| EXP | 1 | 35 | 4 | 80 | EXP | 1 | 21 | 1 | 52 |
| LN | 2 | 34 | 5 | 79 | LN | 2 | 21 | 2 | 52 |
| X**Y | 1 | 35 | 8 | 76 | X**Y | 2 | 21 | 1 | 52 |
| Control Data 6400 | (48 bit mantissa) | | | | SDS Sigma 7 | (24 bit mantissa) | | (56 bit mantissa) | |
| EXP | 1 | 47 | | | EXP | 4 | 20 | 8 | 48 |
| LN | 2 | 46 | | | LN | 4 | 19 | 4 | 50 |
| X**Y | 7 | 41 | | | X**Y | 8 | 15 | 8 | 46 |
| G.E. 225 | (30 bit mantissa, FIZMOP system) | | | | Univac 1107 | (27 bit mantissa) | | (54 bit mantissa) | |
| EXP | 3 | 27 | | | EXP | 2 | 25 | 4 | 50 |
| LN | 12 | 18 | | | LN | 6 | 21 | 7 | 47 |
| X**Y | 10 | 20 | | | X**Y | 6 | 21 | 10 | 44 |
| G.E. 645 | (27 bit mantissa) | | (63 bit mantissa) | | Univac 1108 | (27 bit mantissa) | | (60 bit mantissa) | |
| EXP | 1 | 26 | 14 | 49 | EXP | 2 | 25 | 8 | 52 |
| LN | 4 | 23 | 4 | 59 | LN | 6 | 21 | 6 | 54 |
| X**Y | 1 | 26 | 14 | 49 | X**Y | 8 | 19 | 9 | 51 |

M = maximum number of bits in error. N = minimum number of correct significant bits.

We will show presently that accuracy in exponentiation depends very heavily on the accuracy in the calculation of the exponential function. Note, however, that even with a good exponential function, as is apparently the case in the single precision CDC 6400 and the original IBM 360 libraries, the exponentiation routine can still be in error by two to three significant decimal places or more. Also note that the exponentiation routines corresponding to our methods as well as the single-precision routine on the G.E 645 display primarily round-off error in these tests.

*Error analysis*

There are two major types of error in any function subroutine. The first is transmitted error, i.e., error due to small errors in the arguments. If we assume

$$z = f(x)$$

where $f(x)$ is differentiable, then

$$\delta z \approx x \frac{f'(x)}{f(x)} \delta x \tag{1}$$

where

$$\delta z = \Delta z/z \approx dz/z \tag{2}$$

denotes the relative error in $z$, and $\Delta z$ denotes the absolute error in $z$. It is clear that the transmitted error, $\delta z$, depends solely on the inherited error, $\delta x$, and not on the subroutine. The second type of error is generated error, i.e., that error generated by the computational process. This includes both errors due to truncating an infinite process at some finite point and roundoff errors.

Even infinitely precise subroutines have no control over inherited error. Therefore, in designing subroutines we assume there is no inherited error and seek to minimize the generated error.

Now let us consider the logarithm-exponential method for exponentiation. We use the relation

$$x^y = c^w, \qquad x > 0, \tag{3}$$

where

$$w = ys$$

and

$$s = \log_c(x).$$

From (1) and (2), and recalling our assumption that $\delta x = \delta y = 0$, we see

$$\Delta w = y \Delta s$$

where $\Delta s$ represents only the generated error from the logarithm computation.
If

$$u = c^w,$$

then

$$\delta u = \ell n \, c \, \Delta w + \delta G(w) \tag{4}$$

where $\delta G(w)$ denotes the generated relative error from the exponential computation. For good exponential routines $\delta G(w)$ affects only the least significant one or two bits of $u$. Thus, the relative error in the exponentiation is essentially proportional to the *absolute error* in $w$. Clearly, we want to minimize $\Delta w$ as it appears to the exponential routine.

There are two major contributions to this error: the generated error from the logarithm calculation, and the finite word length of the computer. The second is by far the more important of the two. Suppose the floating-point mantissa of the calculator contains $2t$ significant bits, but $w$ is of the order of $2^t$. Then the floating-point representation of $w$, the argument to be passed to a standard exponential routine, may have a rounding error as large as $2^{-t}$, i.e., $\Delta w \approx 2^{-t}$. Consequently, $u$ may be accurate to only about $t$ bits independently of the accuracy of the logarithm calculation. This is the reason some of our tests found inaccurate exponentiation even though the logarithm and exponential routines appeared to be reasonably accurate.

## *A new approach*

There are at least two alternatives to the traditional computation. One is to resort to "overkill" by carrying out the traditional computation in a higher precision arithmetic. This is expensive in time; it is easy to do for single-precision routines, but difficult for double precision routines. (Is this the approach on the G. E. 645?) The second alternative is to raise the status of exponentiation routines. At the moment they are considered to be secondary routines which call upon the primary routines for the exponential and logarithm. We propose that they become primary, self-contained routines with possible secondary entry points for the exponential and logarithm.

If we accept this major reversal in philosophy, we free the computation of several restrictions. For example, we need not pick $c = e$ in Eqs. (3) and (4), but can make the choice $c = 2$ which appears most natural for a computer, and which introduces the factor $\ell n2 = .69315$ in Eq. (4). This permits us to obtain extra significance in the results of the logarithm computation, as we shall shortly see, and to retain this significance throughout the remainder of the calculation.

The first implementations of the algorithm we will outline were programmed using single-precision fixed-point arithmetic to do single-precision exponentiation on both the CDC 3600 and the IBM 360 computers. Because neither computer allows efficient double-precision fixed-point arithmetic, the algorithm has to be modified to use double-precision floating-point arithmetic to do double-precision exponentiation. So that the presentation will not be too abstract, we will present basically the algorithm as used on the IBM 360 in double-precision. Modifications for single-precision floating point or fixed point versions, or for other machines should be obvious.

We first reduce the range over which the logarithm must be approximated. Let

$$x = 2^k \cdot m, \qquad 1/2 \leq m < 1,$$

and choose

$$b = n/16$$

and

$$a = 2^{-n/16},$$

$n$ an odd positive integer less than 16, such that

$$x = 2^{k-b}\, m/a$$

where

$$|\log_2(m/a)| \leq 1/16.$$

Then

$$s = \log_2(x) = s_1 + s_2$$

where

$$s_1 = k - b,$$

$$s_2 = \log_2\left(\frac{1+z}{1-z}\right),$$

$$\left(\frac{1+z}{1-z}\right) = \frac{m}{a},$$

and

$$z = \frac{m-a}{m+a}.$$

Since $z$ is quite small ($|z| \leq .022$), $s_2$ is easily computed to full floating-point accuracy using a low order rational approximation, or even the first few terms of the Taylor series, provided $z$ is computed accurately. (A little extra care is necessary at this point in base 16 floating-point but we will not go into the details here.) Since $x$ is assumed to be exact, $m$ is exact and we can achieve full precision in $m$-$a$ by breaking the constant $a$ into two parts such that

$$a = a_1 + a_2$$

to the precision desired and such that the exponent on $a_2$ is much less than that on $a_1$. Then the computation

$$m - a = (m - a_1) - a_2$$

will retain the low order bits of $a$. Normal floating-point can be used for the rest of the evaluation of $z$.

Note that by carrying $s_1$ as one floating point number, and $s_2$ as another, we have rather painlessly achieved a logarithm accurate to well beyond usual working precision. Since $|s_2| \leq 1/16$, the *absolute error* in $s$ is now about $2^{-4}$ times the normal relative error in floating point. Careful multiplication of $s$ by $y$ will minimize the crucial quantity $\Delta w$. At this point, the usefulness of fixed-point arithmetic with the extra significant bits in the representation of a number is apparent. When such arithmetic is not available, as we have assumed is the case, it is necessary to arrange the floating-point computations to achieve the extra significance at minimal cost. This is done as follows.

Let us say we *reduce* a number $z$ when we write it in the form

$$z = z_1 + z_2$$

such that $z_1$ is the integer part of $16z$. Essentially, then, $s$ is already in reduced form. We compute the exponent $w$ in reduced form by writing

$$y = y_1 + y_2,$$

where $y_1$ and $y_2$ are the double-precision representations of the most significant and least significant halves of $y$ respectively, and forming the products $s_1y_1$, $s_2y_1$, and $sy_2$. Each of these quantities is again reduced and the results combined to form the reduced

$$w = w_1 + w_2.$$

Now $w_1$ is of the form

$$w_1 = \ell + j/16$$

where $\ell$ and $j$ are integers. We then finally compute the exponential value

$$u = 2^{\ell} \times 2^{j/16} \times 2^{w_2}. \tag{5}$$

Since $|w_2| \leq 1/16$, a Taylor series computation of the exponential is quite efficient, although we used rational Chebyshev approximations. The quantities $2^{j/16}$ can be carried in a table. In fact, if Eq. (5) is rewritten as

$$u = 2^{\ell+1} \times 2^{(j-16)/16} \times 2^{w_2}$$

and the quantities $2^{-n/16}$ are tabulated, the same table can be used for the constant $a$ needed in the logarithm computation. This dictates the form of the earlier decomposition of $a$ into $a_1$ and $a_2$. Clearly $a_1$ should be the value of $a$ correctly rounded to working precision while $a_2$ becomes a positive or negative correction term.

TABLE II—Random argument tests on conventional double-precision X**Y on IBM 360/75

| Argument Range | | Frequency of Bit Errors No. of bits in error | | | | | | | | | | | | Max. Rel. Error | RMS Rel. Error |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| x | y | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | other | | |
| (1/16,16) | (−4,4) | 272 | 467 | 405 | 371 | 240 | 197 | 47 | 1 | 0 | 0 | 0 | 0 | 1.25E−15 | 3.65E−16 |
| $(2^{-16},2^{16})$ | (−16,16) | 78 | 123 | 153 | 168 | 247 | 377 | 321 | 294 | 195 | 44 | 0 | 0 | 8.82E−15 | 2.70E−15 |
| $(2^{-32},2^{32})$ | (−8,8) | 80 | 109 | 131 | 152 | 216 | 288 | 295 | 234 | 241 | 120 | 86 | 48 | 5.08E−14 | 9.60E−15 |
| $(2^{-64},2^{64})$ | (−4,4) | 57 | 95 | 115 | 126 | 161 | 215 | 293 | 352 | 303 | 192 | 82 | 9 | 2.68E−14 | 6.97E−15 |
| $(2^{-8},2^{8})$ | (−32,32) | 59 | 90 | 115 | 109 | 199 | 312 | 406 | 343 | 253 | 107 | 7 | 0 | 1.40E−14 | 4.02E−15 |
| (1/16,16) | (−64,64) | 60 | 96 | 110 | 128 | 196 | 275 | 318 | 318 | 281 | 167 | 48 | 3 | 1.95E−14 | 5.73E−15 |

Average execution time for (x, y) random in (0, 1) = 195 μsecs.

TABLE III—Random argument tests on self-contained double-precision X**Y on IBM 360/75

| Argument Range | | Frequency of Bit Errors No. of bits in error | | | | | Max. Rel. Error | RMS Errror Rel. |
|---|---|---|---|---|---|---|---|---|
| x | y | 0 | 1 | 2 | 3 | 4 | | |
| (1/16,16) | (−4,4) | 1301 | 677 | 22 | 0 | 0 | 2.22E−16 | 6.24E−17 |
| $(2^{-16},2^{16})$ | (−16,16) | 1206 | 759 | 35 | 0 | 0 | 2.22E−16 | 6.11E−17 |
| $(2^{-32},2^{32})$ | (−8,8) | 1314 | 667 | 19 | 0 | 0 | 2.22E−16 | 5.81E−17 |
| $(2^{-64},2^{64})$ | (−4,4) | 1350 | 634 | 16 | 0 | 0 | 2.21E−16 | 5.44E−17 |
| $(2^{-8},2^{8})$ | (−32,32) | 1097 | 812 | 89 | 2 | 0 | 2.22E−16 | 6.31E−17 |
| (1/16,16) | (−64,64) | 872 | 823 | 250 | 52 | 3 | 2.22E−16 | 6.94E−17 |

Average execution time for (x, y) random in (0, 1) = 180 μsecs.

Since the last two factors in $u$ are each less than unity in magnitude, and the $2^{t+1}$ factor affects only the floating-point exponent, we see that the construction of $u$ from its factors is a stable process. Note that the error $\Delta w$, hence by Eq. (4) the error $\delta u$, now depends, primarily on the magnitude of $y$. Using Eq. (4), and noting that we have gained an extra four bits in our calculation of $s$, we see that $y$ must be greater than roughly 32 before the inaccuracies in $w$ become large enough to greatly affect $\delta u$. To verify this point, and to provide an in-depth comparison of our method and of the traditional computation, we have subjected our routine for the IBM 360 and the original IBM routine to a full certification as described in references one and two. The results, for identical tests, are presented in Tables II and III.

One final word about the fixed point version o' this algorithm. In fixed point, the extra bits over t' normal floating point manitssa length are already available. As we have indicated, the decomposition of $a$ and $y$ and the reduction of $s$, $w$, etc. are no longer necessary. This constitutes a savings in storage as well as in the number of instructions to be executed.

But no matter which approach is taken, the fixed point or the floating point, the self-contained routine can be expected to be competitive timewise with the traditional routine because we have saved the overhead of linking with other subroutines. All three of the self-contained programs we have written are actually faster than their traditional counterparts. The price is paid in terms of storage. This price can be minimized by incorporating entries for the exponential and logarithm routines into the exponentiation routine, thus eliminating separate routines for the former.

## ACKNOWLEDGMENTS

We would liketo acknowledgethe assistance of J. Boyle, P. Businger, L. Fosdick, C. Hammer, J. Pagels, R. Royston, H. C. Thacher, Jr., J. F. Traub and the computing centers at Argonne, Bell Telephone Laboratories, University of Illinois, Northwestern University, Univac, and the University of Notre Dame for their assistance in the tests reported in Table I.

## REFERENCES

1 W J CODY
*Performance testing of function subroutines*
Proc SJCC 1969 759-763

2 N CLARK W J CODY K E HILLSTROM E A THIELEKER
*Performance statistics of the Fortran IV(H) library for the IBM system/360*
Report ANL 7321 Argonne Natl Lab 1967

# DCDS digital simulating system*

*by* H. POTASH, A. TYRRILL, D. ALLEN,
S. JOSEPH, and G. ESTRIN

*University of California*
Los Angeles, California

## INTRODUCTION—SIMULATION SYSTEMS

*To see a world in a grain of sand*
*And a heaven in a wild flower,*
*Hold infinity in the palm of your hand*
*And eternity in an hour.*
—*William Blake*

This article is concerned with the problems of digital simulation and describes methods used in the Digital Control Design System (DCDS)[1] for the simulation of digital structures. The paper is divided into five parts:

- A short introduction to DCDS, its structure and purposes.
- A discussion of simulation techniques, entities and attributes.
- The DCDS pseudo machine simulator.
- The pseudo machine program.
- A simple example of a DCDL program.

### *DCDS, its structure and purposes*

The Digital Control Design System (DCDS) was developed at the University of California at Los Angeles to aid in the design and architecture of computer systems. The design system operates under the following assumptions:

1. A set of basic building blocks whose properties are known is available.
2. An instruction set or task assignment for the computer system is defined along with cost and performance constraints.
3. Using his experience and intuition, the designer generates an ensemble of modules. These modules form the system's building blocks which the designer believes will perform the stated functions effectively.

Given the above (1—3), the digital system must be describable to a design aid system. The designer then needs a language, its translator, and an operating system with the following properties:

4. The set of functions to be performed can be described.
5. The building blocks, their interconnection, and their place and function within the ensemble can be described.
6. A computer program can generate a fabrication description of control modules capable of going through a sequence of states necessary to have the system perform the above functions. The designer may specify synchronous or asynchronous control systems.
7. A simulator can accept the descriptions in (4) and (5), and the sequence description generated in (6), and produce measures of accuracy and performance.
8. If the performance of the ensemble is "good", the description of the computer system is in such form that it may be fed into a more de-

* This research was supported in part by the Atomic Energy Commission AT(11-1) Gen 10 Project 14, and the Office of Naval Research, Information Systems Branch, N00014-67-A-0111-0016.

tailed design process. If not, the designer may alter his architecture.

To satisfy the above needs, Digital Control Design Language (DCDL) has been implemented as part of design automation research being conducted at the University of California at Los Angeles.[2-5] A compiler for DCDL has been implemented for the SSD SIGMA 7 using a META 5 compiler writing system.[6,7] The DCDL compiler is currently also being implemented for the IBM/360.

The DCDL system illustrated in Figure 1 contains two compiler processors written in META 5, a pseudo machine (which is the subject of this paper) written in FORTRAN IV and the machine language, and two control implementation modules written in FORTRAN IV. The input processor is a DCDL syntactic analyzer; (1) this program translates the digital system description (example in Part IV) into an interpretive code used by the pseudo-machine for simulation of the described hardware. The second META 5 processor (2) produces a numerical code which is then transformed into a binary control program and a fabrication description of a control subsystem for the computer system being designed. The implementation specifications for the wiring of the control matrices are produced by the two FORTRAN IV programs (3,4). Control modules implied by microprograms have their wiring lists automatically generated by the Control Matrix Processors, in DCDS. The hardware construction of the control processor is then effected by using a set of one or more similar building blocks (Control Matrix Building Blocks), according to wiring specifications given by DCDS.

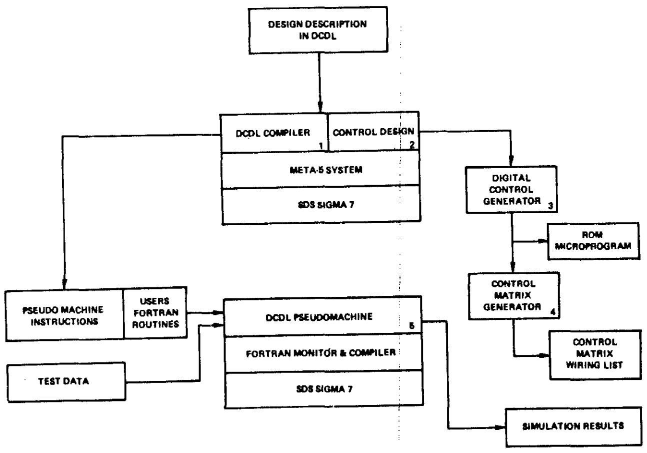

DESIGN DESCRIPTION IN DCDL
DCDL COMPILER 1
CONTROL DESIGN 2
META-5 SYSTEM
SDS SIGMA 7
DIGITAL CONTROL GENERATOR 3
ROM MICROPROGRAM
CONTROL MATRIX GENERATOR 4
CONTROL MATRIX WIRING LIST
PSEUDO MACHINE INSTRUCTIONS
USERS FORTRAN ROUTINES
TEST DATA
DCDL PSEUDOMACHINE 5
FORTRAN MONITOR & COMPILER
SDS SIGMA 7
SIMULATION RESULTS

Figure 1—DCDS system flow chart

The software module described in Part III is a pseudo machine (5) in charge of executing simulation runs. The pseudo machine is composed of a combination of FORTRAN IV and machine language subroutines. The simulation runs are designed to check test cases in order to assess the validity of a described design as well as to calculate its estimated execution time.

DCDS is designed to analyze asynchronous as well as clocked systems, with the former posing a special problem: dynamic reevaluation of variables. Any time a logical variable is changed, the system must, as a consequence of this change, reevaluate any other variable which is a function of the changed variable. This process must continue until no further "consequential changes" occur.

DCDS's capability to dynamically reevaluate variables allows the designer to describe his system using the same logical equations and timing relations which he uses to implement it. Programming in a form (see Part IV) which is highly related to the actual hardware provides for a system directly used by the designer eliminating the programmer as a "middle-man". This direct correspondence also makes the DCDL program an up-to-date documentation of the system designed. The syntax analyzer accepts a description which images the hardware and translates this description into simulation code. Thus the designer is freed from the tedious job of programming the structure of the model required—a process sometimes more time-consuming than building a hardware prototype and testing it on the bench.

The Digital Control Design Language (DCDL) is built as a cluster of three main sublanguages: a language intended for expressing Boolean equations and time relations; a microprogramming language; and an algorithmic language. DCDL uses FORTRAN as the algorithmic sublanguage. The user may choose any one of the three sublanguages to describe any of the parts or modules in the described design. The logical and microprogramming sublanguages use the same declarations and access the same variables by their names. The execution statements of sublanguages and their syntactic formats differ and one cannot combine statements of different sublanguages. Thus DCDS provides the user with a powerful means of expression, since he can select the most convenient and expressive form from among the three sublanguages to describe a hardware module.

### *Entities and attributes in simulation systems*

For our observations herein, we consider the simulation of a system to be the modeling and associated measurement of a system by a STRUCTURE in which EVENTS occur in TIME according to a set of RULES. Thus there are four sets of basic elements which must be dealt with in simulation:

STRUCTURES, EVENTS, TIMES, and RULES

Different simulation methods neglect one or more of these sets (e.g., time independent models). Any one of the four sets may be selected as primary entities and the others treated as attributes of that set.

One may choose to consider an analytic closed form solution to be a simulation of a real system. In this case, the process of simulation becomes a transformation. Assume for example the transfer equation for an electronic circuit. Both *internal events* (voltages and currents in the individual elements) and *structure* (topology of the circuit) may be neglected and one manipulates the set of *rules* (i.e. Kirchoff's law and Ohm's equations) to produce a transfer function which gives the *output events* as a function of *time* and *input events.*

Thus whenever the rules are considered to be the main entities, then either an analytic transformation or an algorithmic procedure is used for simulation. The type and form of the information transferred into the simulation system as well as the simulation systems themselves vary from one another depending upon which of the four sets was chosen as the main set of entities. Due to these differences, different languages or input rules are used to describe the simulated system to the software package designed to perform the simulation.

The following examples of different programming structures will serve to illustrate the previous discussion.

Main Entities—EVENTS

Examples of programming structures:

SIMULA [8], GASP [9], SIMSCRIPT [10], [11], [12], [13], GPSS [14].

A simulated system is described by an event flow chart. The programming systems above use input language formats suitable for the description of events in such a form.

Main Entities—RULES

Examples:

NASAP [15], LISA [16], Boolean Analyzer [17].

The input to circuit analysis programs like NASAP and LISA or to the Boolean Analyzer is in table-form which either explicitly gives the set of rules (Boolean equations) or gives a table that implies a unique set of rules (Kirchoff's and Ohm's equations for the circuit).

Main Entities—STRUCTURES

Examples:

LOGIK [18], Weather Simulation Program [19].

Partial Differential Equation Simulation [20].

The input format is any form suitable for describing the physical or hierarchical structure of the simulated system.

## Modeling and approximations

After the selection of the entity and attribute relations, the next step for simulating a system is to decide what can be approximated and how the selected approximations can be done. The choice of what to approximate can be categorized as:

a. making certain entities (inputs) constants; for example t = 0 in time independent modeling.
b. neglecting parts of the attributes; for example in simulation of partial differential equations by Monte Carlo methods, the field constants are calculated for only a small number of selected field points in the structure.
c. modifying the set of rules; the use of difference equations to solve partial differential equation problems is an example of modifying the rules. For a different example of rule modification, consider a simulation program simulating another program on a digital system. The purpose of the simulated program is to execute a matrix inversion in which the inversion is performed on a 2×2 part of the matrix instead of the entire n×n array. In this case, the system rules may be modified to obtain fast simulation time for a simualtion that "takes the system through the motions" without obtaining the actual numerical result. Thus for such approximations, one may simulate the system faster than real run time.

Event directed simulation can be expected to be faster than structural simulation since structure simulation has to go through all possible events in the system, while event simulation takes the system only through the prescribed events. This is, of course, also the main pitfall of event simulation; it does not point out events that might occur in the system but are unforeseen by the programmer.

*DCDS pseudo machine simulator*

A computer module in DCDL may be described by its structure (LOGIC), by the set of events that it controls (PROGRAM), or by the algorithmic rules (SIMULATE). In order to perform this task, the DCDS pseudo machine simulator operates as an algorithmic simulator by calling on the FORTRAN programs; as a structure simulator when simulating a logical structure (operating from the Call Stack); or as an event simulator when processing a microprogram. The *Program Stack* (see Figure 2) operates the sequence of events generated by the control microprogram. The *Call Stack* operates all the logical details occurring in the logical structures forced by the control events.

The DCDL event simulation is limited to operations within a logical structure. The events that are generated by the control as time moves forward, forces the simulator to follow all consequences of the events within the described logical structure. For example, the event simulator may directly order (by executing an instruction in the program stack) transfer of data to register A. All the other consequences of this action (i.e., all the outputs of gates whose input is A) are simulated from the Call Stack (structure simulation).

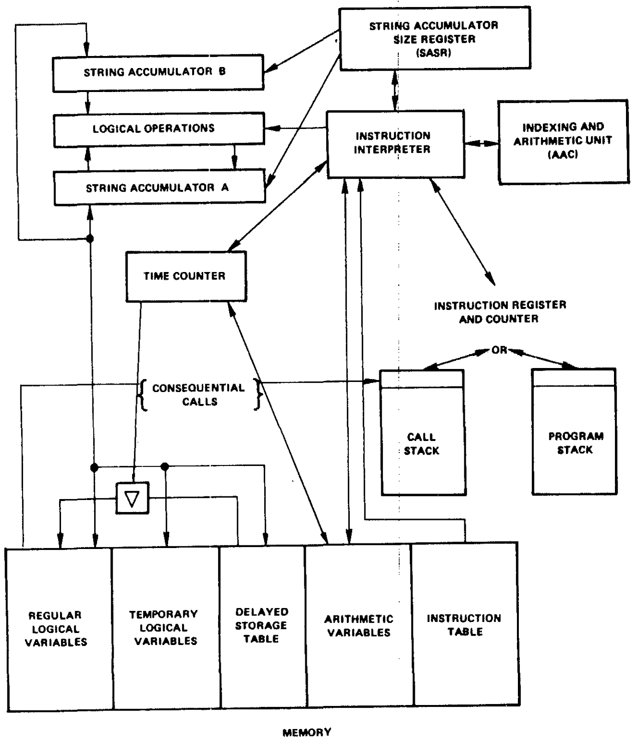


Figure 2—Pseudo machine structure

*The pseudo machine program*

A pseudo machine processor is a program written in machine language or higher level language for the machine on which one performs the simulation runs. In the present implementation on the SIGMA 7 this program is written using FORTRAN and assembly language.

The process in which the translation is separated from the simulation allows one to write the translator program independently of the machine in use. The separation of the compiler program and the pseudo machine program allows independent debugging and changes in each. Modifications in DCDL and its compiler are done by changing the META 5 compiler program. FORTRAN changes in the pseudo machine provide for changes in simulation methods as well as insertion by the designer of other features expressed in FORTRAN to capture event information relevant to one design or another.

Thus, by the process of programming in DCDL and by translation one obtains:

A. Documentation of the design;
B. A check on the consistency and completeness of all logical variables and all logical functions;
C. Automatic implementation of control sections;
D. Simulation runs for given sets of input data; and
E. The amount of time a certain run will take on the described design.

Following is a discussion specifying the pseudo machine structure and operation codes.

## Instructions, interpretation, addressing, and indexing

This unit contains the following parts (see Figure 2).

(a) Time counter and time registers.
The counter counts simulated execution time. The time registers are used to store time counts of different parallel branches. At a parallel junction, comparison between duration of operation on each branch is made and the highest time count will be the new value of the simulation time counter.

(b) Indexing arithmetic unit.
This unit is capable of fixed point operation

(plus, minus, multiplication, and division) and is used for indexing arithmetic.

(c) Call-stack and Program-stack.

Two push down (LIFO) stacks. One of the elements in the stacks is the operative address; i.e., the address of the instruction to be executed next. The operative address is usually the word at the top of the call-stack. If the call-stack is empty, the operative address is the word at the top of the program-stack.
A control branch to a lower (subordinate) control level (CALL) is instrumented by putting the first address of the lower control level program into the call stack, thus making the call address the operative address. When the lower control level is of type PROGRAM, the address is put in the program stack. The operative address is incremented by 1 after an instruction is executed or the address is replaced by another due to the execution of a branch (a normal branch that occurs within the program being executed).

An exit or return from the subordinate program will cause the stack to pop while a further entry into another subordinate program brings a new address into the call stack. The consequential calls are put into the call stack but their execution is delayed until all the parallel operations have been carried out and then all consequential calls are carried out. Two key words in DCDL indicate parallel structures. *GROUP indicates a set of similar modules operating in parallel and controlled by the same binary control variable (for example, a set of 32 single bit adder modules in a 32 bit binary adder). *PART indicates a set of dissimilar modules operating in parallel under the control of a single binary control variable (for example, shifter and counter in floating point normalization). A *PART may contain simple and nested *GROUPS in which case the whole structure is operating simultaneoulsy under the supervision of a single control variable. The stacks have three points. TOPC (top of the call stack). TOPP (top of the program stack) and OPR (the operative address.)

OPR = TOPP if call stack is empty
OPR = TOPC if call stack is not empty
OPR = TOPC at the time of entry to *GROUP or *PART if executing inside a *GROUP or a *PART.

Consequential calls are intended for the dynamic reevaluation of variables. The STORE instruction invoking the consequential calls puts new addresses of variable reevaluation routines into the Call Stack. This is accomplished according to the following steps:

1. The old and the new value of the variable are compared.
2. The new variable value is stored.
3. If the comparison mentioned above shows a difference between the old and new value, the address of the subroutine that calculated the new value of the dynamically dependent variable is put into the Call Stack.
4. The address of the next instruction is the address on the top of the Call Stack. Thus, if there were any consequential calls, they would be executed prior to the completion of the execution of the subroutine that invoked those consequential calls.

When there are no more changes in the values of the variables, the instructions proceed to the end of the reevaluation routine, which contains RETURN as the last instruction. The RETURN instruction pops the Call Stack sending the program to finish operations in the routine which invoked the consequential calls.

The process of dynamic reevaluation will stop only if the variable values and the logical functions are consistent. Assume the following statements:

$A = \wedge\ (B,C);$
$D = \vee\ (A,E);$
$B = \neg\ D;$

with initial conditions $A = 0$, $B = 1$, $C = 0$, $D = 0$, $E = 0$. This set of relations and values is consistent. Now consider that the variable C is changed to one. The new set up of variables and relations is inconsistent and the reevaluation of variables will not reach a steady state. Each reevaluation will put a new address in the call-stack.

A change in operation occurs once an address is put into location n in the stack. The pseudo machine prints an error message which is followed by the names and values of variables partaking in a STORE instruction. This process continues allowing the program to put addresses in the next ten slots of the call stack. When the execution calls for storing an address at n+11 the call stack is cleared (TOPC = 0) and the operative address is taken as the instruction on top of the program stack. This debug feature allows the program to check for logical inconsistencies without getting into an infinite loop or having to stop simulation runs.

5. Delay table. The result of a logical transformation specified in DCDL can be effected directly or after a specified time, for example in the statement

   A = 'DELAY(3)' & (C, D, E): OP1;

   the transformation &(C, D, E) is performed if control variable OP1 is activated, but the content of A will be changed only three time units later.

To facilitate translation of the delay modifier, the pseudo machine contains a delay table. An entry into the delay table contains three parts: variable name variable's new value, time of exit.

| | | |
|---|---|---|
| Variable name | Variable value | Exit time |

Each time the time counter is incremented, all time of exit entries into the delay table are checked, and the entries with a time of exit matching the time counter activates a store operation storing the new value in the appropriate variable, invoking consequential calls if such are present.

## Logical manipulating accumulators

The pseudo machine contains two string accumulators, A and B. The machine performs the operations of AND, OR and EQUAL between the respective bits of the string accumulators and the result is stored in string accumulator A. The current size of both string accumulators is given by the content of String Accumulator Size Register (SASR).

All operations are performed on words of the same size. Calling an operand of the wrong size causes an error message printout and the machine goes to the next instruction. An exception to this occurs when the operand is of size one bit. In this case, the one bit is extended to a word that contains all zeros or all ones of the size indicated by the SASR. A special instruction sets the size of the string accumulator (i.e., the content of SASR) thus setting the size of all following logical operations.

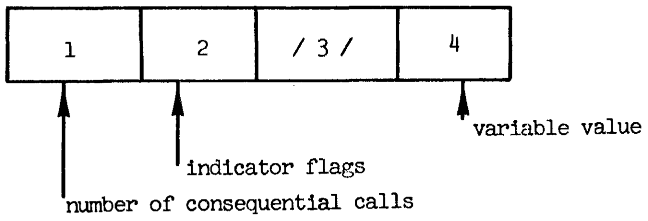
## Data Blocks

Data blocks have different lengths and contain binary arrays. A binary array can possess up to three dimensions. *Only* a single bit or a binary word string can be addressed in the blocks. Each data block contains a two word header containing the variable name followed by the structure described below.

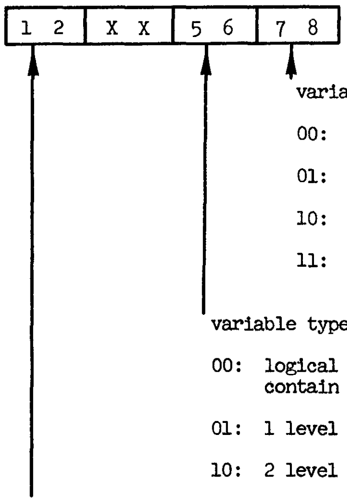
## Storage for a Single Bit

The storage blcok for a single bit is one word (four bytes) plus a word for each consequential call. A consequential call occurs when a variable A is a dynamic function of a variable B. B forms the input to the gate, the output of which is A. When B is changed, a consequential call causes the pseudo-machine to reevaluate the variable A. Thus, the storage location of variable E contains the addresses of sets of instructions which will reevaluate all variables which are dynamically dependent on the variable B.

The single bit storage words format



| | |
|---|---|
| Byte 1: | number of consequential calls invoked by a change in the stored binary variable. |
| Byte 2: | this byte contains indicators for high bit position, number of dimensions of the logical variable, and variable type. Each indicator occupies two bits. |



variable dimension

00: bit variable

01: one dimension array (word)

10: two dimensional variable

11: three dimensional variable

variable type

00: logical point, the variable does not contain memory

01: 1 level storage, declared as *RS

10: 2 level storage (clocked)

position of the high order bit

00: the high order bit is the most significant bit (leftmost bit)

01: the high order bit is the least significant bit (rightmost bit)

Byte 3: not used
Byte 4: variable value

The following words (if any) contain the consequential call address in byte 3 & 4 and its directive in byte 1.

Byte 1: consequential call type (directives)

011: calls on any change in the variable

001: consequential call, only if the variable changes from 0 to 1

010: consequential call, on the change of the variable from 1 to 0

Ixx: consequential call of an entry to a PROGRAM, put a new address on top of program stack (operation on the last 2 bits same as above).

## One dimension array storage

In a one dimensional binary array storage, the first word contains the range and type of the stored variable. The following words contain the binary variable and then the consequential calls (if any).

Format:

| 1 | 2 | 3 | 4 | |
|---|---|---|---|---|
| | | | | variable storage |
| cc directive | cc address | | | |
| | " | | | consequential |
| | " | | | call |
| | " | | | address |

First word: Byte 1: number of consequential calls
Byte 2: variable dimension, high order bit position and variable type (same as for bit storage)
Byte 3: lowest subscript of variable
Byte 4: size of variable.

The second word through the nth word $\left(n = \frac{\text{word size} + 1}{32}\right)$ contain the value of the binary word. If the variable is a clocked F/F, the amount of space for variable storage is doubled and each bit has two storage locations, primary and secondary.

The last set of words contains consequential call addresses and their directives.

## Two dimensional binary storage

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| 5 | 6 | ///// | ///// |
| | | | |
| cc directive | cc address | | |
| | " | | |
| | " | | |
| | " | | |

A two dimensional arrangement contains at least 3 words. The first 2 words are used for bookkeeping in the same format as the 1 dimensional arrangement, with byte 5 indicating the lowest value of the second subscript, and byte 6 indicating the range of the second subscript.

**Three dimension**

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| | | | |
| cc directive | cc address | | |
| | " | | |
| | " | | |
| | " | | |

In a three dimensional arrangement, byte 7 indicates the lowest value of the third variable and byte 8 indicates the range.

**Arithmetic variable storage**

The third entity stored in pseudo memory is a block of 256 arithmetic variables used for indexing and address manipulations.

**Temporary logical variabls**

The memory contains a block of 256 one dimensional logical temporary variables, each one 128 bits long.

**Pseudo machine instruction set**

Most of the pseudo machine instructions closely resemble general purpose computer instruction lists. The main exception is that the addresses of logical variables contain the variable address as well as bit and word indices.

In the following paragraphs we will discuss specific instructions which are unique to the DCDL pseudo machine and will give the reader more insight into DCDS simulating programs.

A pseudo machine logic instruction is contained in a 64 bit word (eight bytes).

As implemented on the SDS Sigma 7, the most common format of the pseudo-machine logic instruction code contains

a. operation code (one byte)
b. operation code modifiers (one byte)
c. operand address (two bytes)
d. three address subscripts and a set of subscript tags.

The actual operand address is a function of the main address (i.e., array address), the three subscripts, and the subscript tags. The main address corresponding to the name of the data block (i.e., the name of the variable). The subscript tags indicate whether the subscripts are to be used directly, indirectly, or by word size.

Each index byte has a two bit tag. The interpretation of the tag is:

If the tag is 00, this subscript is not currently effective. For example, in A(1, 3), A is a two dimensional array and the third index is not used.
If the tag is 01: The subscript is indicated directly by the numerical content of the corresponding subscript byte.
If the tag is 10: The subscript is given directly; i.e., the corresponding number is the location of an indexing word in memory.
If the tag is 11: It is used for word variables and the word is the entire range of this subscript.

The following section contains pseudo machine instruction examples from the set of pseudo machine instructions.

**Store with invoked consequential calls**

STDC a): a ← A, Call Stack ← consq (a)
If there is a difference between (a) and A, all the consequential call addresses associated with (a) are put into the call stack. To avoid redundant operation, a duplication of the address already inside the call stack will not be inserted; i.e., when two or more successive operations request the same consequential call this mechanism sets the operation such that the call will be executed only once. When the receiving variable (a) is a clocked element (two storage levels) both levels change to match the content of A.

**Store in secondary level**

SSEC (a): $(a_1) \leftarrow A$

Stores into first level of a clocked storage element (a clocked element has two storage levels). This instruction does not initiate consequential calls.

**Secondary to primary storage level transfer, entire array**

TRANS (a): $(a_2) \leftarrow (a_1)$

Transfers the data from secondary to primary level in clocked memory elements. This instruction initiates consequential calls if consequential call addresses are present and the content of primary and secondary differ.

### Secondary to primary transfer, only designated bit(s)

BTRANS (a): $(a_2) \leftarrow (a_1)$

Instruction execution same as above except transfer is performed only on bit(s) designated by the instruction. Note: consequential calls are not associated with single bits; a change in a variable invokes all consequential calls for the array.

### Delayed storage

DELAY (a), i: DELAY TABLE ← a, i, A

i, the delay count, is put in the second byte of the eight byte instruction (as a modifier). Delayed storage invokes consequential calls when they are associated with the stored variable. The consequential calls as well as storage will be activated after i time units.

### Instruction format

```
'7 C'- -'- -'- -'- -'- -'- -'- -'
 1   2   3   4   5   6   7   8
```

Byte 2: delay count
Byte 3-8: logical variable address

### Delayed secondary to primary transfer

CKDLY (a), i: DELAY TABLE ← a, i

This instruction stores the address and time count in the Delay Table. The variable value does not have to be stored in the Delay Table since it is stored in the secondary register of the variable.

### *PART entry point

PARTIN:

changes the GROUP flag to 1. As long as the GROUP flag is not equal to zero (GROUP $\neq$ 0) the operative address does not change due to the placement of an address in the call stack.

### *PART exit point

PARTOUT:

Turns the GROUP flag to "0" thereby releasing the consequential calls mechanism. Thus, if consequential calls have been involved, within PART this instruction causes the effective address to be the top of the stack and execution of consequential calls to begin.

### *GROUP entry point

GRUPIN, K1, XR:

Loads the value K1 into the arithmetic variable serving as index register (XR).
Increments the GROUP flag by one (GROUP = GROUP + 1).
Format

```
'E 2'X X'X X'- -'X X'X X'- -'- -'
 1   2   3   4 5   6   7   8
```

Byte 4: arithmetic variable serving as index register (XR).
7&8: number (K1) loaded into the index register (XR).

### *GROUP exit point

GROUP, K2, i, n, XR:

(1) Compares K2 with the value stored in the appropriate index register (XR).
If the values are *equal*:
Decrements the GROUP flag (GROUP = GROUP − 1) and proceeds with the execution of next instruction. Note that if GROUP flag is decremented to zero (GROUP = 0) the stack pointer is moved to the highest occupied position POINT-TOP and stored consequential calls are executed.
If the values are *not equal*:
The index register variable XR is changed by 1 or by −1.
The operative address (next instruction address) is changed to the value n.

```
'E 3'- -'X X'- -'- -'- -'- -'- -'
 1  2    3   4   5   6   7   8
```

Byte 2: (i) Incrementing or decrementing value (1 or −1)
4 (XR) Address of index register
5&6: (n) Label of the instruction at the top of the *GROUP loop
7&8: (K2) upper limit of index register.

The operative address cannot change as long as execution is within a *PART or *GROUP (GROUP $\neq$ 0). The consequential calls will be stored in the call stack and evaluated onc the program exists all the nesting of *GROUP and *PART.

**Unconditional branch**

GOTO n:
Unconditional branch to n: the value n replaces the operative address.

**Conditional branch**

GOTC (k)n:

Branch is taken if the logical accumulator A = 0 and k = 0 or A $\neq$ 0 and k = 1. When the branch is taken, n replaces the operative address in the CALL or PROGRAM Stack.

**Call**

CALL n:

Control transfer. The label n is put on top of the call stack making it the new current operative address.

**Return from a substructure**

RETURN:

The instruction causes the call stack to pop making the next label in the stack the operative address.

**Call microprogram controller**

CALP (n):

puts (n) on top of the program stack

**Return from a microprogram**

RETRNP:

Pop the program stack

**Check bit**

CHECK (a)

The instruction contains a bit indicator (byte 2). The bit indicator is compared with a bit in memory addressed by bytes three-eight. If the bits are the same, the result is no operation; if the bits are different, the instruction executes a RETURN.

**Count time**

TIME, n: (Timer) $\leftarrow$ (Timer) + n, Evaluate delay table.

Counts n time units; note that with each count the delay table will be reevaluated and the instruction will activate delayed storage.

**Store timer**

TIMS (n): (n) $\leftarrow$ (Timer)

Stores the content of the timer in n

**Return to time count routine**

TRET:

This instruction pops the call stack then returns control to the timer control subroutine.

**Bring timer**

TIMI (n): (Timer) $\leftarrow$ (n)

Sets the timer according to the value stored in n.

**Set timer**

TIMO n, m, k: (Timer) $\leftarrow$ n, (timer subroutine) $\leftarrow$m, k.

The instruction contains a new initial value for the timer.

**Gather point for parallel branches in a microprogram**

GATHER (b), j, k:

This instruction appears at the gather point of parallel operation. The instruction contains two numbers, j and k, each stored in a two byte location and used for parallel branch count. k contains the total number of parallel branches coming in to the gather point; j contains the number of branches not yet executed. The arithmetic variable b is used to store the maximum operation time on the parallel branches.

```
operation: if j ≠ 0
  a. j ← j − 1
  b. (b) ← MAX ((b), (timer))
  c. Pop the call stack
        if j = 0
  a. j ← k
  b. (timer) ← MAX ((b), (timer))
```

c. (b) ← 0
d. go to next instruction (past parallel gather)

```
'D 4'X X'X X'- -'- -'- -'- -'- -'
 1  2   3   4   5   6   7   8
```

Byte 4: Arithmetic variable storing time count
5&6: value of k, total number of parallel paths
7&8: value of j, number of parallel paths to be executed

## Logical to numerical variable transfer, first word

SINI (n), (v): B ← (v), (n) ← B(0-31)

The content of the logical variable v is loaded into B accumulator. When the rightmost bits of B(0-31) are loaded into the arithmetic variable n. This arithmetic variable is to be transferred into the simulated section. If the size of B is less than 32, zeros will be put into the leftmost bits of the word.

## Logical to numerical variable transfer, additional words

SIN2 (n), k: n←B(32*k to 31+32*k)

This instruction must follow SIN1 or another SIN2 instruction. The instruction transfers the kth word from B to the arithmetic variable n to be transferred into the simulation section.
Format

```
'5 1" - -'X X'- -'X X'X X'X X' XX'
 1   2   3   4   5    6    7  8
```

Byte 2: contains the address of the arithmetic variable
4: k, position of the word in B.

## Numerical to logical variable transfer, first word

SOUT1 (n), (v): B←n, (v)←B, B←0

This instruction transfers the bits of an arithmetic word n into the rightmost 32 bits of B, then stores the content of B in v, and then resets B (the instruction may invoke consequential calls if they are associated with v). Byte 2 contains the arithmetic variable address.

## Numerical to logical variable transfer, additional words

SOUT2 (n), k: B(32*k to 31+32*k)←n.

Loads the content of (n) into the kth word of B. This instruction must be followed by SOUT1 or another SOUT2.

## Call simulation section

CALSIM, n: B←0, CALL simulation section.

Resets B, then activates the FORTRAN or machine language simulation section. n is the number of the subroutine called.

## Error trap

TRAP:

This instruction must follow a conditional branch. The execution of the instruction consists of printing an error message and then following the branch of the previous instruction, even though the branch conditions were NOT satisfied.

*The logic design of a serial adder*

Figure 3 gives the block diagram of a design specification for a serial adder. The adder contains two clocked shift registers, A and B, containing 16 bits each. Other parts of the adder are a four bit counter COUNT, a carry flip flop C, a single bit sum and carry logic, the adder controller AUC, and a PANEL section.

The sum of A and B generated by the adder replaces the content of B. A is connected to perform a cyclic shift such that at the conclusion of the addition it

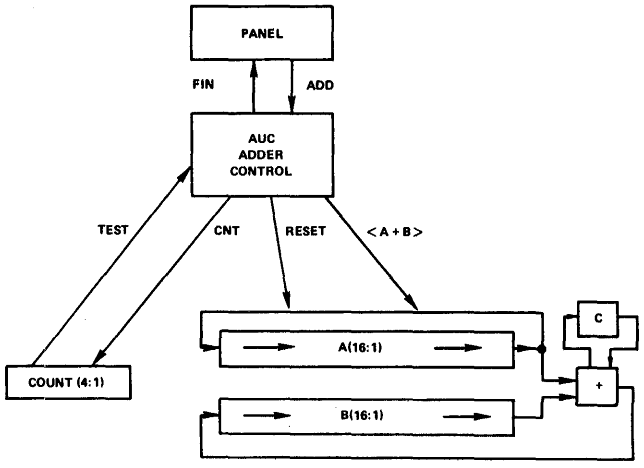


Figure 3—Serial adder

contains its initial value. The sum bit generated at each cycle is stored in position B(16).

## Design Example, Serial Adder

Figure 4 contains a DCDL program specifying the serial adder. The program starts by declaring a UNIT named ADDER at control level #1. The declaration section starting with the key word *DECLARE specifies that the UNIT ADDER receives three control signals (ORDERs) from its supervisor(s). The ORDERs are <A + B>, CNT and RESET. The functions controlled by these ORDERs will be specified later in the LOGIC part of this UNIT.

Other parts declared in this DECLARE section are the 16 bit register A, the 16 bit register B, and the flip flop C. A, B, and C are composed of clocked RS flip flops (type *CRS). The next declaration is a four bit register COUNT constructed from TRIGGER flip flops and a DATA_BUS logic variable TEST. The value of the variable TEST will be specified in the LOGIC part as a logical function of memory elements.

```
*UNIT ADDER, LEVEL=1
*DECLARE
*ØRDER <A+B>,CNT,RESET        ;
*CRS A(16:1) , B(16:1) , C  ;
*TRIGGER CØUNT(4:1)   ;
*DATA BUS TEST      .          ;
*END
*LØGIC
*PART: CNT  ,
 CØUNT(1)%='X1'                ;
CØUNT(2)%=CØUNT(1)             ;
CØUNT(3)%=&(CØUNT(1),CØUNT(2))      ;
CØUNT(4)%= &(CØUNT(1),CØUNT(2),CØUNT(3))      ;
*END
*PART: RESET    ,
C='X0'          ;
CØUNT(*)='X0'          ;
*END
*PART:<A+B>,
A(*)%='CYCLE(-1)' A(*)         ;
B(16)%=|(&(A(1),B(1),C), &(-A(1),-B(1),C),
        &(-A(1),B(1),-C), &(A(1),-B(1),-C))     ;
*GRØUP I=1,15   *SET ,
B(I)%=B(I+1)         ;
*END
C%=|(&(A(1),B(1)),&(A(1),C), &(B(1),C))      ;
*END
 TEST= &(CØUNT(1),CØUNT(2),CØUNT(3),CØUNT(4))       ;
*END
*END ADDER
*UNIT AUC, LEVEL=2
*DECLARE
*ØRDER ADD          ;
*REPLY  FIN        ;
*END
*PRØGRAM
ADD: RESET: A1                  ;
 A1:<A+B>:A2           ;
 A2 :<A+B>,CNT : A3 ;
 A3: *GØ_TØ TEST: (A4,A2)       ;
 A4: *RETURN FIN                ;
*END   *END AUC
*PANEL AAA, LEVEL=3
 *SYSTEM_RESET: A(*)='X7F37', B(*)='X2ECD'; *TIME=0;
*AT_TIME_INTERVAL = 2,WRITE A(*):X, B(*):X, CØUNT(*):X ;
 *START ADD; *FINISH FIN;                        *END AAA
```

Figure 4—Serial adder DCDL program

The declaration section ends with the key word *END.

The logical and control relations in the ADDER UNIT are specified in the LOGIC section which starts with the key word *LOGIC. The LOGIC section contains three PART sections and one direct transfer statement.

The first PART section is controlled by the ORDER_VARIABLE CNT. This section contains the input statments to the four COUNT flip flops. The statements specify that the input to COUNT (1) is a "ONE" ('X1' specifies a one in a hexadecimal format). The input to COUNT (2) is the output of COUNT (1). Similarly the input to COUNT (3) is the AND of (COUNT (1), COUNT (2)) and the input to COUNT (4) is the AND of (COUNT (1), COUNT (2), COUNT (3)).

The first PART section is controlled by CNT clocked transfers (%=) which are associated with the clocked input of the registers' flip flops. The next PART section controlled by the ORDER RESET specifies a direct connection (=) into the clocked variables C and COUNT. Therefore, the PART controlled by CNT changes the clocked input of the COUNT register. The PART controlled by RESET changes the content of COUNT and C using direct set (DC set) and direct reset (DC reset).

The last PART section is controlled by the ORDER_VARIABLE <A + B>. Activated by the <A + B> control variable are the following transformations:

a. The content of A is shifted a cyclic shift by one to the right, the result is stored in A(*);
b. B(16) receives the sum function of A(1), B(1), and C;
c. The GROUP of bits B(1) to B(15) are shifted by one to the right;
d. The carry flip flop C receives the carry which is a function of A(1), B(1) and C.

Note the PARTs containing a clocked transfer refer to double rank clocked elements. Whenever the controlling variable is activated, the specified function (to the right of %=) is stored in the secondary rank of the variable to the left of %=. In the succeeding time unit, a primary secondary transfer is activated.

The last statement in LOGIC is a dynamic specification of the variable TEST as an AND function of the bits of COUNT.

The next UNIT to be specified is the adder controller, AUC. AUC introduces two new variables in its declaration section: an ORDER ADD which it receives from its supervisors, and a reply FIN which it sends back to the supervisors.

The control function of AUC is specified by a microprogram in the PROGRAM section of AUC. The interpretation of the microprogram is as follows:

a. When a controller receives the ORDER ADD, it issues the ORDER RESET. After the default time lapse, two time units, the controller switches to state A1.
b. In state A1, the controller issues the ORDER <A + B>. After two time units, the controller moves to A2;
c. At state A2 the controller issues two ORDERs <A + B> and CNT. The next state is A3;
d. A3 is a conditional branch. If TEST is "ONE", the next state is A4. If TEST is "ZERO", the next state is A2. The GO__TO line is an internal control branch specification which does not require any additional cycle. Therefore the execution time of this line is zero time units;
e. The last microprogram line states that when the controller is in state A4 it issues the REPLY pulse FIN, and returns to its zero state.

The highest controller in the structure is AAA PANEL at level 3. The PANEL specifies the system's initial conditions (placing initial values in A and B) using the SYSTEM RESET statement. The initial condition for the timer is specified by the statement *TIME = 0. The key word *START indicates the initiating variable, and the key word *FINISH is followed by the variable signaling completion. The last statement in PANEL is *END followed by PANEL's label AAA.

### More Complex Structures

The above description has illustrated the use of DCDL to design a simple adder. The language and system have been used to design more complex structures including a multiplier and special purpose logic card tester.[1]

## CONCLUSION

The scope of the DCDS study was limited to systems for which a set of predefined building blocks and a defined structure are present. A total design automation system requires programming tools capable of studying, simulating, and gathering statistics and thereby able to evaluate conjectures about the behavior of structures and sequences of events before the details of the structures and events are known. We hope that further extension of DCDS and further study in simulation and modeling will add the capability to make conjectures based on systems less rigorously defined than DCDS presently requires them to be.

The DCDL implementation by sublanguages which are compiled by META5 allows a simple insertion of other sublanguages designed to study the architectures of systems. The DCDL pseudo machine operates as a FORTRAN based simulator either to describe the simulated system or to augment the pseudo machine instruction set.

## BIBLIOGRAPHY

1 H POTASH
*A digital control design system*
UCLA Dept of Engineering RpT No 69-21 May 1969
PhD Dissertation

2 R L MANDELL
*Tools for the construction of design automation systems*
UCLA 1968 PhD Dissertation

3 R MANDELL G ESTRIN
*A meta-compiler as a tool for design automation*
Proc SHARE Design Automation Workshop 1966
New Orleans Louisiana

4 R A RUTMAN
*LOGIK, a syntax-directed compiler for computer bit-time simulation*
UCLA Masters Thesis Aug 1964

5 K P GOSTELOW
*LOGIK, a system for the computer-aided selection and assignment of electronic modules*
UCLA Rpt No 68-8 March 1968

6 D OPPENHEIM
*The META 5 language and system*
Tech Memo TM-2396/000/01 System Development Corp
Santa Monica Jan 1966

7 D OPPENHEIM D HAGGERTY
*META 5: A tool to manipulate strings of data*
Proc 21st Nat Conf of Association for Computing
Machinery 1966

8 O DAHL K NYGUARD
*SIMULA, a language for programming and description of discrete event systems*
Introduction and User's Manual Norwegian Computing
Center Forskningueien 1B Oslo 3 Norway May 1966

9 P J KIVIAT A COLKER
*GASP—a general activity simulation program*
P2864, RAND Corp Santa Monica 1964

10 B DIMSDALE H M MARKOWITZ
*A description of the SIMSCRIPT language*
IBM Systems Journal Vol 3 No 1 1964

11 M A GEISLER H M MARKOWITZ
*A brief review of SIMSCRIPT as a simulating technique*
RAND Corp RM-3778-PR Santa Monica 1963

12 B HAUSER H M MARKOWITZ
*Technical appendix on the SIMSCRIPT simulation programming language*
RAND Corp RM-2813-PR Santa Monica 1963

13 H M MARKOWITZ
*SIMSCRIPT, A simulation language*
Prentice-Hall Englewood Cliffs N J 1963

14 R EFRON G GORDON

*A general purpose digital simulator and examples of its application: Part I—description of the simulator*
IBM Systems Journal Vol 3 No 1 1964

15 L P McNAMEE H POTASH
*A user's guide and programming manual for NASAP*
UCLA Dept of Engineering Rpt No 68-38 Aug 1968

16 K L DECKERT E T JOHNSON
*User's guide for LISA 360, a program for linear systems analysis*
IBM System Development Division TR 02-432 San Jose July 31 1968

17 M A MARIN
*Applications for the Boolean analyzer*
UCLA Dept of Engineering 1968 PhD Dissertation

18 R A RUTMAN
*LOGIK, a syntax-directed compiler for computer bit-time simulation*
UCLA Masters Thesis Aug 1964

19 Y MINTZ
*Very long term global integration of the primitive equations of atmospheric motion*
Meteorology Monographs Vol 8 No 30 Feb 1968

20 A F CARDENAS
*A problem oriented language and a translator for partial differential equations*
PhD Dissertation UCLA 1968

# Pattern recognition in speaker verification

*by* S. K. DAS and W. S. MOHN

*IBM Corporation*
Research Triangle Park, North Carolina

## INTRODUCTION

There are many ways in which a pattern recognition system may be implemented. In the specific problem of speaker verification,[1,13] a two-class recognition scheme is of interest. A speaker who desired verification of his identity based upon some previously stored characteristics of his speech represents one of the two classes (real), whereas the other class (impostor) encompasses all other speakers.

In implementing such a system, it is convenient, first, to obtain a representation for each of the utterances of interest in the form of a time-frequency-amplitude matrix.[2,3] The conventional method of deriving this representation is by means of a filter-bank analyzer.[2,3] Speech signals are inputted to the analyzer and the outputs of the various filters are sampled and averaged over the appropriate time interval. This process generates a set of short-term average spectra with which to form the time-frequency-amplitude matrix.

Normally, only those components of this matrix which contain significant speaker characteristics need be retained. Identification of such speaker-dependent components is somewhat arbitrary although several guide lines are available.[2,3]

The next step is to regard all the pertinent elements of the above-mentioned matrix as constituting a single vector. Thus, the net result of the previous processing steps is a vector representation for each utterance.

At this stage, several mathematical and statistical tools may be applied appropriately to the data. For example, the vector representation of an utterance may exbihit high dimensionality. For further computational advantage, it is desirable to reduce this dimensionality of the vector. It is also helpful to achieve as much intra-class clustering and inter-class separation as possible. Methods such as analysis of variance,[4] discriminant analysis[5] and mutual information calculation[10] are available for this purpose. The analysis of variance and mutual information methods can be conveniently used even if the initial dimensionality of the vectors is rather high. The disadvantage of these two methods is that each element of the vectors is considered independent of the other elements; this is not desirable since the interrelationships between the elements which may be important for the purpose of speaker verification are completely ignored. On the other hand, while discriminant analysis treats the vectors in multi-dimensional space, thereby preserving the interrelationships, the computation time required may be impractical if the vectors are initially of inappropriately high dimensionality.

Finally, a method for discriminating among the vectors of the real class and of the impostor class is required. This is usually done by means of a reference vector. There are again several alternatives here. For example, it has been pointed out that if a suitable representation for the impostor class is not available, it is possible to derive a reference vector based on the real class data only.[6] But, if the impostor class is properly characterized, Adaline-type linear threshold elements,[7] which attempt explicit discrimination among the real and the impostor classes, may be used to advantage.

There are many other methods for feature selection and reference vector generation.[7,8] Each method has its particular advantages and shortcomings. In addressing the speaker verification problem, it is convenient to use the analysis of variance technique for feature selection and a modified form of the Adaline-type linear threshold device for deriving a reference vector. Previous unreported in-house experimentation has indicated that the two techniques, analysis of variance and mutual information calculation, produce rather similar sets of features. The rationale for using the modified form of the Adeline device will be presented in the next section.

Another important aspect of the pattern recognition problem is to conduct a significant experiment with true data. Too often, for reasons of economy and time limitations, artificially generated data or a very limited quantity of true data is used to perform experiments. Experience has shown that conclusions based on such experimentation are often misleading. A primary achievement of the present experiments is the use of a large true data base. The value of this large data base will be further appreciated in the following sections.

The next section outlines the modification of the Adaline-type linear threshold element. The analysis of variance technique, being a standard tool in statistics, is not treated here. The details of the experimental part are listed in the third section. Finally, some conclusions and observations regarding the whole procedure are made.

## *Theory*

At first, in this section, a brief description of the classical Adeline[7] procedure will be given and some of its shortcomings will be pointed out. Next, a modified form of the above procedure will be developed. In the standard Adaline technique,[7] a reference (or weight) vector $W$ is derived by utilizing vectors from both of the two classes to be recognized, C and $\overline{\text{C}}$. The vectors in the two classes are assumed to be linearly separable.[7] For convenience in describing the technique, the negative of the vectors belonging to $\overline{\text{C}}$ are assigned to C. Next, denote the vectors that are now attributed to C as $Y_1, \ldots, Y_m$, where m is the total number of vectors. The Adaline procedure[7] is an iterative method of determining a weight vector, $W$, such that

$$Y_j \cdot W > 0 \qquad j = 1, 2, \cdots, m \qquad (1)$$

where the operator $(\cdot)$ signifies the inner product operation of two vectors. The iteration process is described by the rule

$$\begin{aligned} W_{k+1} &= W_k \qquad \text{if } Y_k \cdot W_k > 0 \\ &= W_k + Y_k \text{ otherwise.} \end{aligned}$$

Using this weight vector $W$, a new test vector may be classified to C or $\overline{\text{C}}$ depending on whether the inner product of the test vector with $W$ is greater than zero or not. The drawback of this procedure of decision making is that the test pattern vectors belonging to C which would normally produce slightly positive inner products may, in the presence of some noise, lead to negative inner products and be misclassified. Similar statements may be made about the patterns belonging to $\overline{\text{C}}$.

In order to avoid this difficulty, a weight vector $W'$ which satisfies the inequality

$$\begin{aligned} &Y_j \cdot W' > \text{K}|Y_j|^\gamma |W'|^\alpha \\ &\qquad j = 1, 2, \cdots, \text{m} \qquad (2) \\ &\qquad 0 \le \text{K} < \infty, 0 \le \gamma < \infty, 0 \le \alpha < \infty \end{aligned}$$

may be tentatively proposed for classification. Clearly, the advantage of this inequality is that for non-zero K, a dead-zone is created. This zone is symmetric about zero and equal in magnitude to the right-hand side of equation (2). The dead-zone may be designated as an interval of no decision. As a result, some tolerance to noise is provided. A noisy test pattern vector which would otherwise satisfy equation (2) may lead to an inner product lying in the dead-zone, but is unlikely to to be misclassified.

However, only some special cases of equation (2) will really concern us. The cases which will not be of interest are

$$\begin{aligned} &Y_j \cdot W' > \text{K}|W'|^\alpha \\ &\qquad j = 1, 2, \cdots, \text{m} \qquad (3) \\ &\qquad 0 \le \text{K} < \infty, 0 \le \alpha < \infty, \alpha \ne 1 \end{aligned}$$

The reason they are not of interest will now be given. It will be demostrated that a $W'$ satisfying equation (3) may be derived from a $W$ satisfying equation (1) by a simple change in magnitude; thus since $W$ and $W'$ would differ only in magnitude and not in orientation, the classification ability of $W'$ would be identical to that of $W$. (It should be pointed out that if an actual iteration process is carried out to arrive at a weight vector $W$ for equation (1) and a weight vector $W'$ for equation (3), K≠0, the weight vectors are likely to be oriented differently in the multi-dimensional space and would thus lead to different generalizations. The important point to note is the possibility that $W'$ may be oriented in the same direction as one of the possible $W$ vectors.)

Assume that a weight vector $W$ has been found by some means which satisfies equation (1). Denote the minimum of the inner product values indicated in the eft-hand side of equation (1) by $\delta$, i.e.,

$$\operatorname*{Min}_{j} Y_j \cdot W = \delta > 0$$

The postulate will be shown to be true by deriving a scalar constant $S > 0$ such that the weight vector

$$W' = SW$$

will satisfy equation (3). Since the above operation on $W$ changes its magnitude only and not its direction, the minimum of the inner products $Y_j \cdot W'$ still occurs at the same value of j (more than one value of j may produce the minimum value, but this fact is of no concern to the present development). Let this value of j be designated as j′. Thus,

$$Y_j' \cdot W = \delta$$

Then multiplying both sides of the above equation by S yield

$$Y_j' \cdot SW = S\delta$$

or

$$Y_j' \cdot W' = S\delta$$

In order to achieve equation (3), it is necessary to find an S such that

$$\begin{aligned} S\delta &= K|W'|^{\alpha} \\ &= KS^{\alpha}|W|^{\alpha} \end{aligned}$$

Rewriting the above equation,

$$S = \left( \frac{K|W|^{\alpha}}{\delta} \right)^{1/1-\alpha}$$

which is the suitable value for satisfying equation (3). Note that S cannot be determined for $\alpha = 1$.

If among all possible cases of equation (2) the cases given in equation (3) are not considered, the equation of interest is either an inequality in the form of equation (3) with $\alpha = 1, \gamma = 0$.

$$Y_j \cdot W' > K|W'| \qquad j = 1, 2, \cdots, m \qquad (4)$$
$$0 < K < \infty$$

or, the general equation (2) with $\gamma \neq 0$,

$$Y_j \cdot W' > K|Y_j|\gamma|W'|^{\alpha}$$
$$j = 1, 2, \cdots, m \qquad (5)$$
$$0 < K < \infty, 0 < \gamma < \infty, 0 \leq \alpha < \infty$$

Equation (4) has another interpretation. If this equation is written as

$$Y_j \cdot \frac{W'}{|W'|} > K \qquad \begin{array}{l} j = 1, 2, \cdots, m \\ 0 < K < \infty \end{array}$$

it is apparent that the inner products of the weight vector, normalized to unity, and the vectors $Y$ are computed in this algorithm. This implies that the only way the weight vector can affect the value of the inner products is by changing its direction in space and not by changing its magnitude. This fact is of considerable interest since it has already been demonstrated how a simple change in magnitude of a weight vector satisfying the simple equation (1) can make the new weight vector satisfy the more complex equation (3), even though the generalization property remains unchanged.

The use of equation (4) is advocated in this paper. In the following paragraphs, however, an approach to the more general equation (2) will be considered first. Substitution of suitable values for the different parameters of equation (2) will then realize the results for equation (4). It will be found that a bound for the convergence rate may not always be obtained for the approach adopted in this paper.

The procedure parallels the proof for the standard Adaline technique.[7] The iteration method is defined by the equations

$$\begin{aligned} W'_{j+1} &= W'_j \text{ if } Y_j \cdot W'_j > K|Y_j|\gamma|W'_j|^{\alpha} \\ &= W'_j + (\mu/|Y_j|^{\beta})Y_j \text{ otherwise;} \end{aligned}$$
$$0 < \mu < \infty, 0 \leq \beta < \infty$$

where the constants $\mu$ and $\beta$ have been incorporated for further generality.

Following standard convention, a reduced training sequence $\hat{Y}_1, \hat{Y}_2, \ldots, \hat{Y}_k, \ldots$ and a reduced weight vector sequence $\hat{W}_1, \hat{W}_2, \ldots, \hat{W}_k, \ldots$ are formed[7] and the following discussion is based on these sequences. The initial gain vector is assumed to be zero, $\hat{W}_1 = 0$. Then,

$$\hat{W}_{k+1} = \mu \sum_{i=1}^{k} \frac{\hat{Y}_i}{|\hat{Y}_i|^\beta} \tag{6}$$

Taking the dot product of the solution vector $W'$ (which is unknown, but is assumed to exist) with both sides of the above equation yields

$$\hat{W}_{k+1} \cdot W' = \mu \sum_{i=1}^{k} \frac{\hat{Y}_i \cdot W'}{|\hat{Y}_i|^\beta}$$

From equation (2)

$$\hat{Y}_i \cdot W' > 0$$

Let

$$\underset{i = 1, 2, \cdots, k}{\text{Min}} \frac{\hat{Y}_i \cdot W'}{|\hat{Y}_i|^\beta} = A_m$$

Then,

$$\hat{W}_{k+1} \cdot W' \geq k\mu A_m$$

But, the Cauchy-Schwarz inequality states that

$$|\hat{W}_{k+1}|^2 \geq \frac{(\hat{W}_{k+1} \cdot W')^2}{|W'|^2}$$

Therefore,

$$|\hat{W}_{k+1}|^2 \geq \frac{k^2 \, v^2 \, A^2{}_m}{|W^1|^2} \tag{7}$$

This is a lower bound for the magnitude of $\hat{W}_{k+1}$. Another line of reasoning will give an upper bound as well.
Since

$$\hat{W}_{j+1} = \hat{W}_j + \frac{\mu \hat{Y}_j}{|\hat{Y}_j|\beta}$$

it follows that

$$|\hat{W}_{j+1}|^2 = |\hat{W}_j|^2 + \mu^2 |\hat{Y}_j|^{2(1-\beta)} + 2\mu \frac{(\hat{Y}_j \cdot \hat{W}_j)}{|\hat{Y}_j|^\beta}$$

Therefore, using $\hat{Y}_j \cdot \hat{W}_j \leq K|\hat{Y}_{j}\gamma| \hat{W}_j|^\alpha$.

$$|\hat{W}_{j+1}|^2 - |\hat{W}_j|^2 \leq \mu^2 |\hat{Y}_j|^{2(1-\beta)} + \frac{2\mu K |\hat{Y}_j|\gamma |\hat{W}_j|^\alpha}{|Y_j|^\beta}$$

The above inequality for j = 1, 2, $\cdots$, k may be added to obtain

$$|\hat{W}_{k+1}|^2 \leq \mu^2 \sum_{j=1}^{k} |\hat{Y}_j|^{2[1-\beta]} + 2\mu K \sum_{j=2}^{k} |\hat{Y}_j|^{\gamma-\beta} |\hat{W}_j|^\alpha \tag{8}$$

From equation (6),

$$\hat{W}_j = \mu \sum_{i=1}^{j-1} \frac{\hat{Y}_i}{|\hat{Y}_i|^\beta}$$

Hence,

$$|\hat{W}_j| \leq \mu \sum_{i=1}^{j-1} |\hat{Y}_i|^{(1-\beta)}$$

Let

$$B_M = \underset{i = 1, 2, \cdots, j-1}{\text{Max}} \quad |\hat{Y}_i|[^{(1-\beta)}]$$

Then,

$$|\hat{W}_j| \leq \mu(j-1) \, B_M$$

Also, let

$$C_M = \underset{j = 2, 3, \cdots, k}{\text{Max}} \quad |\hat{Y}_j|^{\gamma-\beta}$$

Therefore, the last term in the right-hand side of equation (8) may be written as

$$2\mu K \sum_{j=2}^{k} |\hat{Y}_j|^{\gamma-\beta} \, |\hat{W}_j|^\alpha \leq 2\mu K \sum_{j=2}^{k} C_M \mu^\alpha (j-1)^\alpha B_M^\alpha$$

Thus, equation (8) leads to

$$|\hat{W}_{k+1}|^2 \leq \mu^2 \, k B_M^2 + 2\mu^{1+\alpha} \, K B_M^\alpha \, C_M \sum_{j=2}^{k} (j-1)^\alpha$$

Since,

$$\sum_{j=2}^{k} (j-1)^\alpha = \sum_{j=1}^{k-1} j^\alpha \leq \int_0^{k-1} (j+1)^\alpha \, dj = \frac{k^{1+\alpha}}{1+\alpha}$$

it follows that

$$|\hat{W}_{k+l}|^2 \leq \mu^2 k B_M^2 + \frac{2}{1+\alpha} \mu^{1+\alpha} K B_M^\alpha C_M k^{1+\alpha} \quad (9)$$

which gives the upper bound on the magnitude of $\hat{W}_{k+1}$. Combining equations (7) and (9),

$$\frac{k^2 \mu^2 A_M^2}{|W'|^2} \leq |\hat{W}_{k+1}|^2 \leq \mu^2 k B_M^2 + \frac{2}{1+\alpha} \mu^{1+\alpha} K B_M^\alpha C_M k^{1+\alpha} \quad (10)$$

which must be satisfied if a solution vector exists.

At this point, it is necessary to substitute suitable values of the various parameters to gain further insight For example, if the parameters are set as

$$\beta = 0, \mu = 1, \text{ and } K = 0$$

the standard bounds described in the literature[7] for conventional Adaline-type devices are obtained. For the experiments reported in this paper, the parameters are

$$\beta = 0, \mu = 1, K > 0, \gamma = 0, \alpha = 1$$

Substitution of these parameter values in equation (10) leads to

$$\frac{k^2 A_m^2}{|W'|^2} \leq |W_{k+1}|^2 \leq k B_M^2 + K B_M k^2 \quad (11)$$

where

$$A_m = \underset{j = 1, 2, \ldots, k}{\text{Min}} \hat{Y}_j \cdot W'$$

$$\text{and } B_M = \underset{j = 1, 2, \ldots, k-1}{\text{Max}} |\hat{Y}_j|$$

Thus, the above inequality leads to

$$k \frac{A_m^2}{|W'|^2} \leq B_M^2 + k K B_M$$

The left- and right-hand sides of this inequality have been plotted in Figure 1. It is clear when the two straight lines intersect, as in case (1),

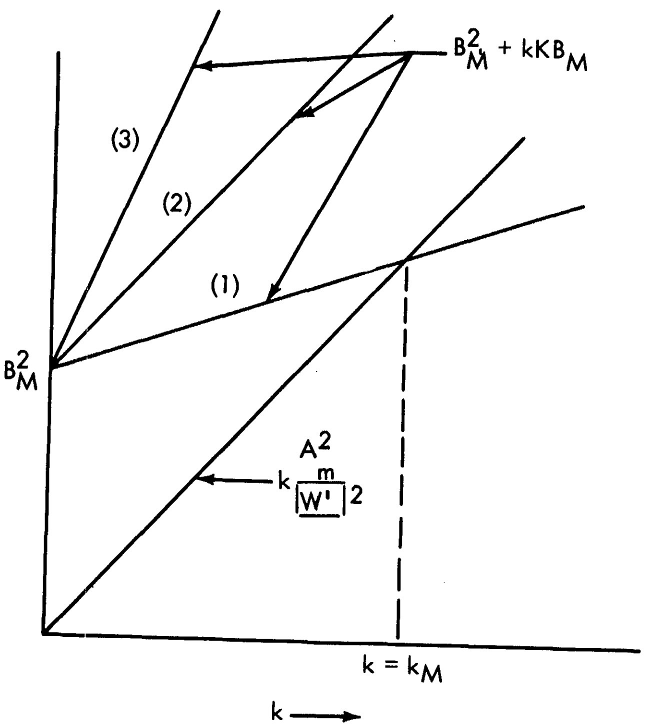


Figure 1—Cases showing presence and absence of upper bound

$$\frac{A_m^2}{|W'|^2} > KB_M,$$

a definite upper bound on the number of steps $k = k_M$ exists, provided, of course, that the solution vector exists. On the other hand, for cases (2) and (3) where

$$\frac{A_m^2}{|W'|^2} \leq KB_M,$$

no such upper bound exists and convergence of the procedure is not guaranteed.

As in standard literature,[7] the bound is not useful in estimating how many steps will be required in a given situation, since it depends on the knowledge of a solution vector $W'$.

It has been shown that the algorithm in equation (4) it desirable because it forces the gain vector to change its direction in space; a simple change in magnitude cannot help in satisfying the inequality of equation (4). At the same time, the possibility of obtaining a section in a finite number of steps exists. In the next solution, this algorithm is the basis for some experiments with real data.

*Experimental results*

The nature of speaker verification allows one to perform experiments which are fairly well controlled. Since most speaker verification applications provide cooperative users—individuals desiring verification—it is possible to require each user to utter a particular phrase. The phrase can be designed to carry a maximum of speaker-dependent information. The choice for the experiments being reported here was "Check Available Terminals." Each speaker included in the test was asked to utter this and four other such phrases in a predefined but randomized order, interspersing each utterance with an utterance-labeling task to prevent interaction between adjacent phrases. Recordings of these utterances were made in an acoustically treated room using a wide-band recording system. A boom-mounted microphone and headset combination assured constant microphone placement. Each subject was asked to speak in a normal voice and a level adjustment made to provide approximately the right input signal level to the tape recorder. It was felt that these rather idealized conditions would allow evaluation of optimum verification performance. In addition, in certain applications the real data may approach this idealized high quality.

TABLE I—Filter bank specifications

| *Filter Number* | *Center Frequency Hz.* | *± db Bandwidth* |
|---|---|---|
| 1 | 188 | 250 |
| 2 | 459 | 250 |
| 3 | 715 | 250 |
| 4 | 969 | 250 |
| 5 | 1220 | 250 |
| 6 | 1472 | 250 |
| 7 | 1725 | 250 |
| 8 | 1975 | 250 |
| 9 | 2225 | 250 |
| 10 | 2475 | 250 |
| 11 | 2725 | 250 |
| 12 | 2991 | 290 |
| 13 | 3300 | 330 |
| 14 | 3659 | 390 |
| 15 | 4083 | 460 |
| 16 | 4586 | 550 |
| 17 | 5194 | 670 |
| 18 | 5954 | 860 |
| 19 | 6932 | 1110 |
| 20 | 8203 | 1450 |

In total, utterances from 118 male speakers were used. Fifty of these were arbitrarily assigned as "reals" and 100 utterances of each phrase were collected from each speaker over about a five-to-ten-week period. Each of the other speakers was assigned to the "impostor" class; each uttered each of the five phrases 20 times, all at one time.

The analog recordings were digitized using the hardware shown in Figure 2 and Table I. It consisted primarily of 20 bandpass filters covering the range of center frequencies 188 Hz to eight kHz. The lower frequency filters had 250 Hz. bandwidths while the higher frequency filters were somewhat broader. A 20-ms. sampling interval was employed. The output of each filter was rectified and integrated over each sampling interval. The value of this integral was converted logarithmically into a four-bit value spanning a 32-dB. range. Only two other pieces of hardware were used—an automatic level control (ALC) and a fundamental frequency detector. The former maximized use of the full dynamic range of the A/D conversion system. Further, to allow reconstruction of the original absolute signal level, the value of the gain of this ALC circuit was digtzied for each sampling interval. The fundamental frequency (pitch) detector also passed a digital estimate of the pitch period to the computer for each sampling interval. Otherwise, this pitch information would have been unavailable because of the width of the bandpass filters. Smith has described the pitch determination method used.[15]

The remaining experimental steps were executed through programming. It was felt that implementing most of the system by software and using a general-purpose hardware analyzer would maximize the flexibility of the system. Even greater flexibility could be obtained by simply sampling the analog speech waveform and storing digitized samples, but the quantity of the data to be processed would be prohibitive. In

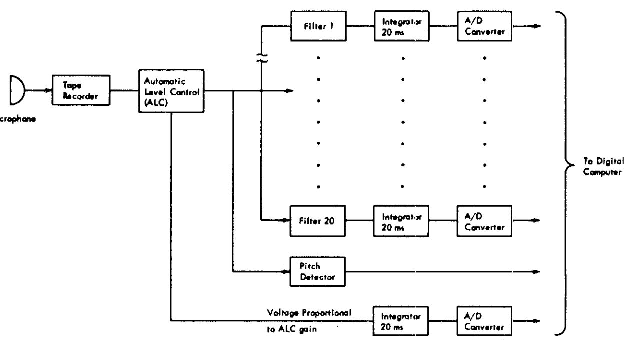


Figure 2—Functions of analyzing hardware

total, approximately 13 hours of analog recordings of the phrase, "Check Available Terminals," were processed.

The first step of utterance-processing, segmentation, was a speech-recognition process which would operate with good reliability over a large population of speakers because the phrase to be recognized was known. This step automatically eliminated improperly spoken or digitized utterances. It also served a time-alignment function, allowing comparison of like sounds from utterance to utterance and from speaker to speaker.

Ten points in time were found for each utterance. Each segmentation point was defined by a precise set of acoustic rules which will not be given here. The points were given the following symbols which correspond roughly to the standard orthography of the words·

checK aVaILaBle TERMinal$S_1$ $S_2$

($S_1$ = onset of S;

$S_2$ = end of S.)

The segmentation rules were determined by the following iterative process. A group of ten speakers was selected arbitrarily and programs were designed to segment their utterances properly. Accuracy of segmentation was verified by studying digital spectrogram patterns of each of the utterances. Once designed, the rules were tested on another arbitrary set of ten speakers. The accuracy of segmentation was improved by accounting for factors manifested in these new speakers. Once the rules seemed sufficiently accurate, in terms of testing on new utterances by this combined set of 20 speakers, these rules were evaluated on an independent group of 20 speakers. Performance appeared consistent; that is, no significant segmentation problems were apparent in this new set of speakers and the segmentation programs were considered complete. Space does not permit detailed description of the final segmentation rules. Roughly speaking, they involved the following functions: voicing detection, frication detection, total signal power, and second formant frequency. Consideration of these functions and the known context of a fixed phrase permitted quite accurate segmentation over a broad population of speakers.

As a preliminary rule, all utterances were required to have defined locations for all ten segmentation points. This restriction resulted in ten percent of the phrases being rejected. Phrase rejection which implies no decision by the machine as to speaker identity should be contrasted with speaker rejection. Most applications would be less sensitive to unnecessary phrase rejection than speaker misclassification. Furthermore, the phrase-rejection rate could be reduced substantially if later stages of recognition were designed to operate on a partially segmented utterance.

The next phase, feature extraction, used a segmented utterance for input and produced a vector of features. For this set of experiments, determining the features was a two-step process. First, a large set of "proposed" features was selected. This choice was based upon previous research by the authors and their colleagues, as well as on published results of experiments involving human and automatic speaker identification.[1,4,8,9,11] Second, the list of features was shortened for economy of implementation. The "goodness" criterion used to determine whether or not to include a particular feature was the F-ratio of analysis of variance.[4]

A detailed list of the proposed features would be too lengthy to include here; instead, the general types of functions employed will be described. A complete description of the features is given elsewhere.[14] The most common function was an integration of the power in one or more filters over a number of time samples. To perform this integration, the log power values determined by the hardware were converted to a linear scale, summed, and then reconverted to log scale. This had the effect of simulating the same type of analyzer with broader filters and longer integration intervals. Three "bandwidths" were chosen for integration: a single filter, a band of several filters, roughly approximating a single formant region, and the entire set of filters, corresponding to the power in the original signal during the 20-ms sampling interval. Three intervals of integration were also used: a short period of two to four time samples centered at a segmentation point, medium-length intervals extending from one segmentation point to the next, and long intervals encompassing several segmentation points. Most of the combinations of these integration regions were employed at each segmentation point.

In order to detect finer differences between utterances, a section of each utterance was subjected to "time normalization." The time-frequency matrix of filter values from the sample labeled "V" to that labeled "L" was "stretched" or "shortened" by linearly interpolating the sampled output of each filter integrator to provide a fixed number of samples. Various integrals like those described above were determined during this time-normalized section also.

Programs were written to estimate approximate formant frequencies and amplitudes as well. Formants are characterized by amplitude maxima in the frequency spectrum and are the result of the transfer function of the vocal tract.[2] There is reason to believe that consistent differences exist among various speakers in absolute formant frequencies and detailed formant transitions from sound to sound, even though the approximate motions are the same from talker to talker. These would reflect an interplay between individual structural and behavioral differences.

These various functions resulted in a total of 405 proposed features. It was obvious by their design that they were not independent, neither functionally nor statistically, but no logical basis was available to select independent features that would be good, *a priori*, for speaker verification. The second step of reducing the feature set employed analysis of variance, ranking the 405 features according to their F-ratio. This measures a quantity proportional to the variance of the speakers' means divided by the mean of each speaker's variance. Such a measure has the desirable properties of invariance to translation and scaling. No measure of feature dependent was calculated. The rank orders were tested for consistency across different speaker populations. The rankings were determined for two different groups of 25 speakers each and rank correlation coefficients were calculated.[12] It was determined that the F-ratio was a consistent measure of relative feature worth when computed over a set of 25 speakers. All of the experiments to be reported used the same feature set, the best 200 features being determined by a composite ranking based on 50 speakers. Details of the ranking are given elsewhere.[14]

Provision had to be made for features that sometimes did not exist or for which an estimate of value did not exist. For example, for certain portions of some utterances the system was unable to determine, adequately, pitch frequency or some formant frequency. This phenomenon will occur to some degree in all feature-extraction systems. A missing feature value poses interesting theoretical problems in the design of a decision method. Should one estimate a value for it on the assumption that the feature really did exist but the system was not sophisticated enough to determine its value? Or should the feature really be presumed missing in the original signal and the utterance considered in a special manner indicating that it is not like utterances in which the feature appeared to exist? Sebestyen[8] addressed these questions in relation to probablistic decision methods, but another approach seemed needed for the non-parametric Adaline technique used here. One possible good approach would be to determine the relative frequency with which each feature was missing in both the real speaker training data and that of the training impostors. During recognition, a value would be substituted for each missing feature which favored neither the real nor the impostor class. Such a value could be the mean of the feature value averaged over both real and impostors. The fact that it was missing would be realized by changing the *a priori* probabilities of the two classes in accordance with the previously stored relative frequencies. Thus, if a real speaker consistently had a feature missing during training, and that same feature was missing during recognition, the recognition threshold would be shifted in favor of accepting the utterance as that of the real speaker.

In the experiments reported here a simpler strategy was employed because of the relative infrequency of missing features. A mean value was retained for each feature that was ever missing from the real speaker's training data. During recognition, if one of these features was missing, the stored mean value was used as an estimate of the missing feature. If a feature was non-existent during recognition but always existed during training, the utterance was ignored entirely. Almost all utterances ignored in this way were impostor utterances and recognition performance would probably not be degraded significantly if each of these utterances was classified as being that of an impostor, but these statistics were not calculated. Approximately four percent of the recognition impostor set of utterances were ignored in this way.

The adaptive linear decision algorithm described earlier was used for all experiments described here. Preliminary experiments were performed to determine a good value for K, the relative training threshold. It was determined that $K = 5$ provided a good tradeoff since convergence was obtained in a reasonable amount of time and higher values of K significantly increased training time with little improvement in generalization performance.

In order to perform training, the set of real utterances was stored in memory. The larger set of impostor data resided on direct access storage. The algorithm proceeded through the data, obtaining utterances alternately from the real and impostor sets. When the end of either set was reached, selection began again at the first of the completed set but continued from wherever it happened to be in the other set. The method provided rapid convergence since the algorithm was always presented with a member of the "other" class after adapting to the first class. A "pass" was defined to be one complete loop through the longer of the two lists of utterances—in our case, the impostor set.

The training data consisted of approximately the first 50 utterances of the real speaker being tested and nine from each of the 29 impostors. These rather arbitrary numbers were the result of practical factors, such as program running time, storage space, and the total number of speakers and utterances available. Further experiments have indicated that generalization results do not depend strongly on the exact amount of impostor training data unless one significantly reduces the number of impostors involved.

The recognition data consisted of the remaining utterances from the real speaker (about 50) and all available utternaces (20 or less) from each of 39 other impostors, Thus, testing generalization of acceptance of the real speaker involved utterances produced by him after producing all of the training data, while generalization of impostor rejection was tested using entirely new people that the training algorithm had never processed.

Computation time on an IBM Sytsem/360 Model 40 was approximately one minute for both each training pass and recognition of 700 utterances.

Table II lists the results of these experiments. The accuracy figure tabulated for each real speaker is the misclassification rate (impostor as real and real as impostor) for the case of the two classes being equally likely. In many applications the *a priori* probabilities would be unequal and the costs associated with the two types of errors would be different, thereby making a statement of a single misclassification probability uninformative. Ignoring the question of cost differences, the distribution of errors was usually such that unequal *a priori* probabilities should allow reduction, or at least no increase, in the probability of system misclassification (both types combined). Figure 3 shows typical distributions of recognition dot products for two real speakers. The probability-density function of the dot product has been integrated from the left for the real speakers and from the right for the set of recognition impostors. The ordinate value corresponding to a particular abscissa value corresponds to the percentage error that would be experienced for that class (real or impostor) if the recognition threshold were placed at that value.

TABLE II—Generalization error over fifty speaker real set (Crossover error rate)

| *Speaker* | *Passes** | *Error (%)* | *Speaker* | *Passes* | *Error (%)* |
|---|---|---|---|---|---|
| 1 | 5 | .3 | 26 | 3 | .2 |
| 2 | 10 | .8 | 27 | 4 | 5.1 |
| 3 | 7 | 1.2 | 28 | 3 | 1.4 |
| 4 | 6 | .0 | 29 | 2 | .0 |
| 5 | 5 | 1.4 | 30 | 9 | .0 |
| 6 | 9 | .0 | 31 | 2 | .6 |
| 7 | 5 | .1 | 32 | 14 | .0 |
| 8 | 4 | .1 | 33 | 2 | .3 |
| 9 | 4 | .7 | 34 | 2 | .0 |
| 10 | 8 | 2.0 | 35 | 5 | .0 |
| 11 | 2 | .2 | 36 | 6 | .4 |
| 12 | 6 | .2 | 37 | 9 | 1.3 |
| 13 | 15** | 2.2 | 38 | 6 | .1 |
| 14 | 5 | .5 | 39 | 6 | .7 |
| 15 | 5 | .0 | 40 | 6 | .7 |
| 16 | 11 | 1.2 | 41 | 8 | 3.1 |
| 17 | 9 | 3.4 | 42 | 4 | .0 |
| 18 | 15** | 2.3 | 43 | 4 | .3 |
| 19 | 11 | .2 | 44 | 4 | 2.3 |
| 20 | 5 | .4 | 45 | 5 | 1.2 |
| 21 | 5 | 1.8 | 46 | 8 | .0 |
| 22 | 3 | .0 | 47 | 4 | 1.8 |
| 23 | 12 | 7.3 | 48 | 3 | .2 |
| 24 | 3 | 2.1 | 49 | 2 | .7 |
| 25 | 4 | .3 | 50 | 4 | .1 |

* Number of passes to reach convergence.

** Convergence not reached by 15 passes, non-converged gain used.

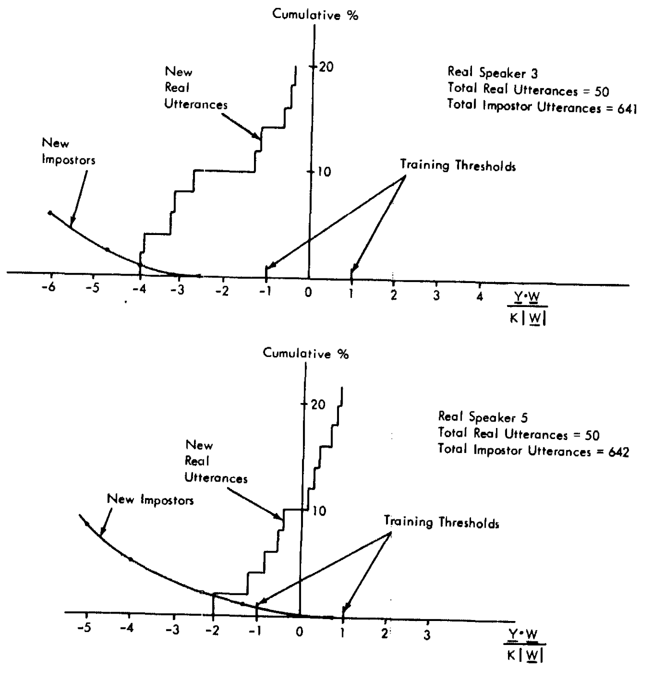


Figure 3—Typical results of generalization tests

The training algorithm was designed to produce an optimum recognition threshold of zero (positive-dot product corresponding to the real speaker, negative to impostors), but the resulting decision function was not symmetrical about the origin. Thus, the accuracy figures in Table II are based upon adjusting the recognition threshold to produce equal misclassification probabilities on recognition data. To obtain an intercept, the step-like nature of the cumulative real distribution was smoothed by linear interpolation. In practice, the recognition threshold must be set in some other way since independent data from the real speaker may not be immediately available. One method might be to set the threshold to produce a fixed rate of impostor

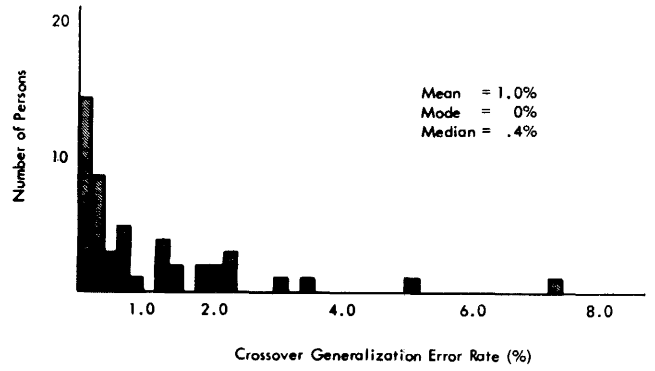


Figure 4—Generalization error histogram

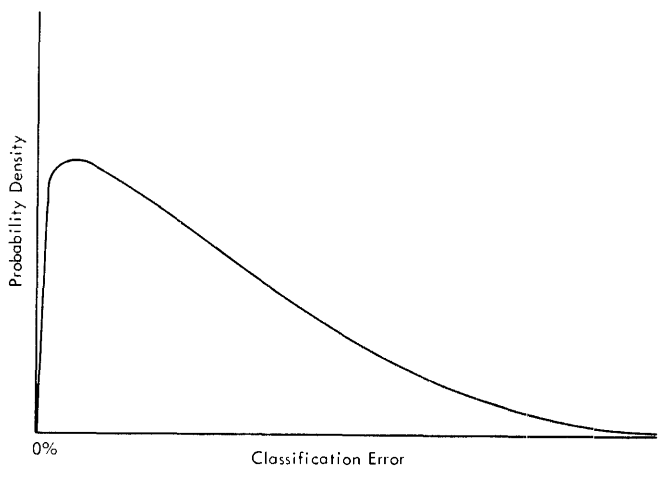


Figure 5—Probable true distribution of accuracy across many real speakers

acceptance (Type II error in statistical terms) and let the real rejection rate be undetermined until the real speaker uses the system a number of times.

Figure 4 shows a histogram of the 50 accuracies in Table II. This distribution bears a resemblance to an exponential form, as might be expected. One would always expect a small percentage of people to have unusually high error rates but no one can have negative error rates; hence, the skewed distribution. If a sufficient number of recognition utterances was available from the real speakers to allow accurate estimation of very low error rates, the true distribution of error would probably look more like Figure 5. This fundamentally imperfect accuracy would result from the inevitable variation in speech patterns with time and because, in the limit of a large enough recognition impostor set, someone would probably be found who is similar to any given speaker, at least within the precision of the features being used.

## CONCLUSIONS AND FURTHER WORK

A more general technique than the conventional Adaline[7] approach has been treated in this paper. The upper and lower bounds of equation (11), applicable to the present method, have been derived from the general bounds of equation (10). These general bounds may be exploited for other applications of equation(2).

In the conventional Adaline method, the iteration process guarantees a solution in a finite number of steps if a solution exists. In the approach adopted in this paper, the iteration process guarantees a solution in a finite number of steps if a solution exists and if this solution satisfies the condition of equation (11).

Since in either of the above two cases the solution vector is not known beforehand, the difference is only a philosophical one. The experiments reported in this paper, however, demonstrate that solution vector can indeed be found in most cases.

The value of the large data base is pointed out again. First of all, this large data base is directed toward an adequate representation for the real and the impostor classes. Even after the data base is divided to conduct independent design and te stexperiments, the above postulate remains largely valid. Also, in many phases of the speaker-verification work (e.g., feature selection), an iterative method is unavoidabla. Thus, once a tentative design is created on some date, the design is tested on a different set of data. If the design shows faults (large error rate), a new design is implemented by using both the former design and the former test data. This new design must now be tested on an entirely different set of data. This type of iterative procedure can only be realized if a large data base is available.

It is felt that the accuracy obtained in the verification experiments is good and that enough people were involved in the test to produce meaningful results. The most comparable previously reported experimental results[1,13] state average accuracies of about ten percent with no provision for "No Decision." Differences in data bases prohibit exact comparison of verification systems. The authors' results cover a significantly larger base of reals than either of the previous experiments.

The authors feel that much of the improvement in accuracy is the result of phrase selection and carefully designed segmentation algorithms but some of the improvement must be attributed to the rather idealized conditions under which utterances were gathered. However, the procedure was automatic once the segmentation program was designed. Further work is being pursued to determine the effect on current results of degrading the signal in both bandwidth and signal-to-noise ratio. Female speakers will also be considered. Improved results are most likely to be obtained through improving segmentation accuracy and flexibility, and the use of more sophisticated features (given better segmentation). It is felt that the present accuracy could be attained with fewer than 200 features by combining dependent features, if storage space presented a significant problem.

## ACKNOWLEDGMENTS

The authors are indebted to their associates in the Speech Processing and other Advanced Technology departments, IBM Systems Development Division Laboratory, Research Triangle Park, N. C., and to consultants Dr. K. P. Li and Dr. D. F. Stanat for invaluable assistance in theory, hardware design and construction, data gathering and pre-processing, and programming. This research is an outgrowth of earlier work performed in association with the Advanced Analog Products department.

## REPERENCES

1 K P LI J E DAMMANN W D CHAPMAN
*Experimental studies in speaker verification, using an adaptive system*
Journal of the Acoustical Society of America Vol 40 Nov 1966 966-978

2 J L FLANAGAN
*Speech analysis, synthesis, and perception*
Academic Press New York 1965

3 C C TAPPERT N R DIXON D H BEETLE JR W D CHAPMAN
*A dynamic-segment approach to the recognition of continuous speech: an exploratory program*
Tech Rpt No RADC-TR-68-177 Rome Air Development Center Griffis AFB N Y June 1968

4 S PRUZANSKY
*Talker-recognition procedure based on analysis of variance*
Journal of the Acoustical Society of America Vol 36 Nov 1964 2041-2047

5 S S WILKS
Math Statistics John Wiley and Sons Inc N Y 1962

6 S K DAS
*A method of decision making in pattern recognition*
IEEE Trans on Computers Vol 18 April 1969 329-333

7 N J NILSSON
McGraw-Hill Book Co N Y Learning Machines 1967

8 G SEBESTYEN
Decision-Making Processes in Pattern Recognition
Macmillian Co N Y 1962

9 G L HOLMGREN
*Speaker recognition, speech characteristics, speech evaluation, and modification of speech signals—A selected bibliography*
IEEE Trans on Audio and Electroacoustics Vol 14 March 1966 32-29

10 L A KAMENTSKY C N LIU
*Computer-automated design of multifont print recognition logic*
IBM Journal of Research and Development Vol 7 Jan 1963 2-13

11 J W GLENN N KLEINER
*Speaker identification based on nasal phonation*
Journal of the Acoustical Society of America Vol 43 Feb 1968 368-372

12 M G KENDALL
*Rank Correlation Methods* Hafner N Y 1962

13 J E LUCK
*Automatic speaker verification, using Cepstral measurement*
Journal of the Acoustical Society of America Oct 1969 to be published

14 W S MOHN
*Statistical feature evaluation in speaker identification*
Dept of Electrical Engineering N C State Univ July 1969
PhD dissertation

15 C P SMITH
*Speech data reduction*
AD-117-290 Clearinghouse for Federal and Scientific
Tech Info 1957

# A hybird / digital software package for the solution of chemical kinetic parameter identification problems

*by* ALAN M. CARLSON

*Electronic Associates, Inc.*
Princeton, New Jersey

## INTRODUCTION

The modern hybrid computer offers many significant improvements over first generation hybrid systems These improvements include:

1. The increased speed of digital computers enabling programs to be written in hybrid FORTRAN without drastically limiting hybrid solution rates.
2. The development of analog/hybrid software (e.g., hybrid simulation languages and analog set-up programs).

The net result of these improvements has been an increase in the scope and complexity of hybrid applications and a reduction in the effort required to program and debug hybrid problems. Unfortunately, the development of hybrid applications software has not kept pace with recent hybrid improvements.

Applications software for purposes of this discussion is defined as an integrated set of digital/hybrid programs capable of solving the majority of frequently occurring problems in a specific applications area. Based on this definition, little or no tangible information is currently available on the practicality of developing hybrid software packages although its benefits are obvious.

In mid-1968, EAI's Princeton Computation Center initiated a development project to determine the feasibility of hybrid applications software. The objectives of the project were to select a frequently occurring application area, develop general purpose software for it, and assess the resultant software based on the above definition, computer economics, ease of use, etc. The objectives of this paper are to present and illustrate the use of the software package developed as a result of the above mentioned project.

The chemical kinetic data analysis problem, which is often referred to as the chemical model building or parameter identification problem was selected as the applications area. Since the software package, which will be referred to as the kinetic data analysis or KDA package, solves chemical kinetic problems via either all-digital or hybird simulations; the question of simulation economics and accuracy was investigated and will also be discussed.

The illustrative problem is the "Monsanto Benchmark Problem" which has been well documented[2,3,6–8] and typifies the chemical kinetic problems the KDA package was designed to solve. This problem requires the determination of twenty-two unknown parameters using thirteen sets of experimental data and a mathematical model requiring the simultaneous solution of seven non-linear differential equations.

## Problem analysis

Referring to Figure 1 the kinetic data analysis problem, which occurs during the initial phases of, say, plant design and economic optimization projects, has three essential, related parts. They are:

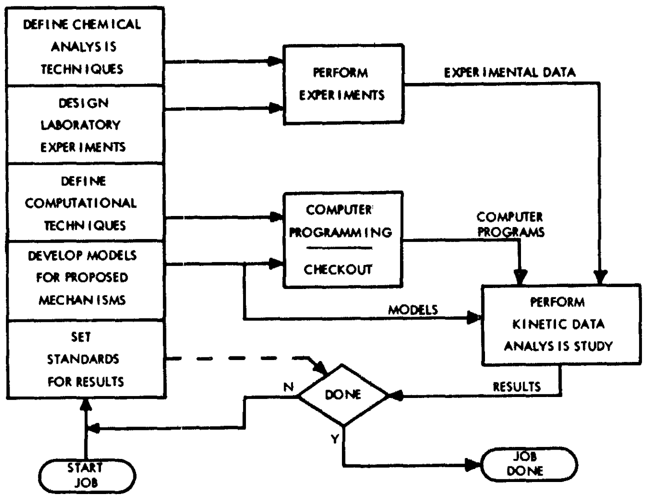

DEFINE CHEMICAL ANALYSIS TECHNIQUES
DESIGN LABORATORY EXPERIMENTS
DEFINE COMPUTATIONAL TECHNIQUES
DEVELOP MODELS FOR PROPOSED MECHANISMS
SET STANDARDS FOR RESULTS
PERFORM EXPERIMENTS
EXPERIMENTAL DATA
COMPUTER PROGRAMMING
CHECKOUT
COMPUTER PROGRAMS
MODELS
PERFORM KINETIC DATA ANALYSIS STUDY
RESULTS
N
DONE
Y
START JOB
JOB DONE

Figure 1—Typical kinetic data analysis flow diagram

1. Performing kinetic experiments to obtain the data necessary to determine the model.
2. Proposing one or more mathematical models representing alternative kinetic mechanisms, chemical reactions, etc.
3. Computational analysis of the proposed models by determining values for model parameter (e.g., rate constants) that minimize the discrepancy between computed and experimental results.

The technology required to design and perform kinetic experiments is available and the initial derivation of mathematical models to simulate these experiments is not generally regarded as a diffiult task. However, the applications software required to evaluate these models is either unavailable, restrictive in a physical sense, or fails to provide the user with an efficient solution to his problem.

The project manager responsible for the solution of a kinetic data analysis problem, based on an impromptu survey, is not interested in becoming deeply involved in programming or underwirting extensive program development studies to solve his problem. With the exception of a few industrial organizations, the computational alternatives at his disposal are not consistent with his interests. The computational alternatives are:

1. Direct Simulation—The classical analog computer or digital simulation language study[10] where the analyst adjusts model parameters in a trial and error fashion. This technique is generally successful; however, it is very time consuming, susceptible to human error, and inefficient except for small problems. The advantage of direct simulation is that it provides the analyst with a great deal of knowledge about the physical behavior of the system being simulated.
2. Parameter Estimation—A variety of digital computer programs that solve kinetic problems using, for example, statistical techniques, line and non-linear least squares, etc. Specific illustrations may be found in a recent article by Lapidus and Bard.[5] Unless the analyst is familiar with these programs and is capable of using them without making major modifications, their utilization creates a number of problems. These problems include:

   A. The mathematical techniques restrict the form of the data or the model, thereby influencing the design of kinetic experiments (e.g., batch-isothermal experiments).
   B. The infrequent use of statistical techniques or lack of a working knowledge of statistics makes it difficult for the user to evaluate program results and equate them to the physical problem.

   Parameter estimation programs do, however, represent a relatively economical means of solving kinetic problems if they can be used efficiently and without major revisions.

3. Parameter Optimization—This technique uses general purpose optimization algorithms (e.g., gradient search) to automate the above mentioned direct simulation technique. Referring to Figure 2, the optimization variables, $\lambda$, which are unknown parameters in the kinetic model, are varied so as to minimize an objective function. The objective function, F, is a scalar quantity representing the error between computed and experimental results which may be obtained using a variety of mathematical relationships (e.g., sum of squares, integral of the absolute error, etc.). As shown in Figure 2, the best current values of the algorithm variables, $\lambda_B$, are those model parameters resulting in the "best fit" between experimental and computed concentration data, $\lambda_F$, when the algorithm can no longer improve the objective function. This technique is:

   A. Theoretically the most general purpose approach to solving kinetic data analysis problems. It may be used in either all-

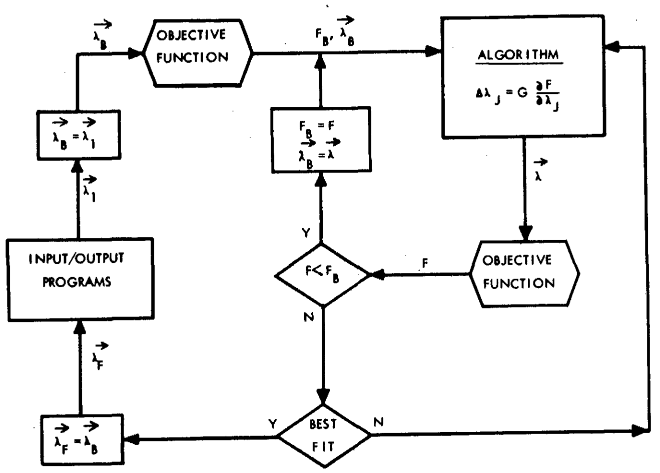


Figure 2—Simplified parameter optimization flow diagram

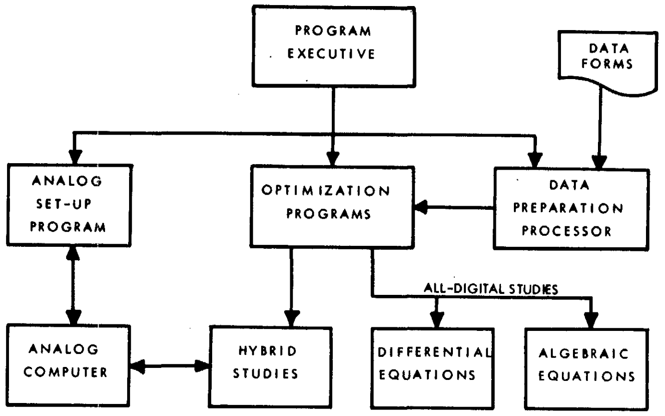


Figure 3—KDA program organization

digital or hybrid simulations and the mathematical forms of the kinetic models and physical systems that can be investigated are not restricted.

B. Not generally used because many organizations do not have access to appropriate software and the development of this software imposes an intolerable financial burden on any one project. In the past, this technique was not widely used due to high digital production costs. The "Parameter Optimization" technique, requires several hundred simulations of individual experiments per optimization run.

The results of the above mentioned survey indicated a significant market existed for general purpose kinetic data analysis applications software if it could produce easily interpretable results, require minimal user participation, and solve kinetic data analysis problems at a reasonable cost using the "Parameter Optimization" technique. These results were used as guidelines for the software development project.

*Software description*

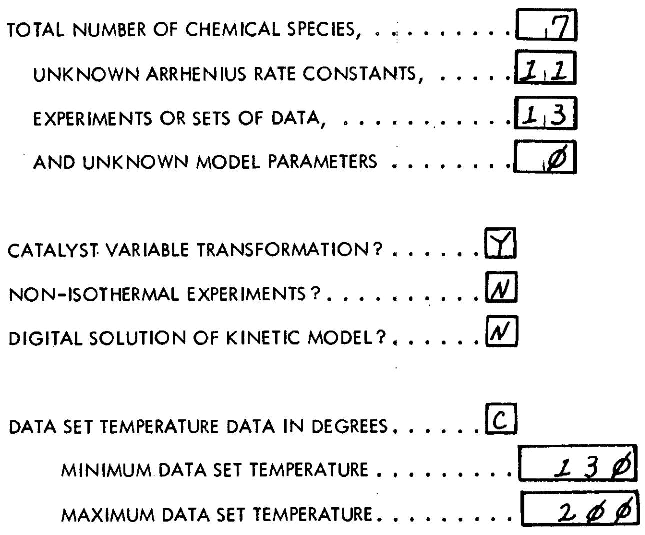
The Kinetic Data Analysis package consists of several digital/hybrid processors whose individual functions and interactions are too complex to describe in this paper. However, referring to Figure 3, the current version of these processors may be visualized as five FORTRAN programs under the control of a Program Executive. The Program Executive restores and executes programs requested by the user, provides the software package with a convenient mechanism to add programs, etc.

The five programs shown in Figure 3 are an Analog Set-Up Program, a Data Preparation Processor, and three optimization programs. The optimization programs are identical with the exception of the mathematical form and/or computer used to simulate the kinetic model or models. These programs, which have identical executive, optimization, and objective function programs are:

1. A hybrid optimization program using the analog computer to simulate kinetic models.
2. An all-digital optimization program for kinetic models requiring the solution of one or more ordinary differential equations.
3. An all-digital optimization program for kinetic models requiring the solution of a set of algebraic equations (e.g., continuous stirred-tank reactor experiments.)

The Analog Set-Up Program is an interactive program used, for example, to static check analog patch panels prior to executive hybrid production runs. Since programs of this type are generally part of the operating system software for a hybrid computer, a description of this program will not be presented in this paper. Subsequent discussions will also exclude the Program Executive, since its function has, for all practical purposes, already been defined. Therefore, the description of the Kinetic Data Analysis package will be limited to the Data Preparation Processor and the optimization programs.

A brief description of how the user interacts and communicates with the software package to solve a kinetics problem will be discussed first to clarify later discussions.



| | |
|---|---|
| TOTAL NUMBER OF CHEMICAL SPECIES, ......... | 7 |
| UNKNOWN ARRHENIUS RATE CONSTANTS, ..... | 11 |
| EXPERIMENTS OR SETS OF DATA, ............ | 13 |
| AND UNKNOWN MODEL PARAMETERS ........ | Ø |
| CATALYST VARIABLE TRANSFORMATION? ...... | Y |
| NON-ISOTHERMAL EXPERIMENTS? .......... | N |
| DIGITAL SOLUTION OF KINETIC MODEL? ...... | N |
| DATA SET TEMPERATURE DATA IN DEGREES...... | C |
| MINIMUM DATA SET TEMPERATURE ......... | 13Ø |
| MAXIMUM DATA SET TEMPERATURE......... | 2ØØ |

Figure 4—Typical KDA data form

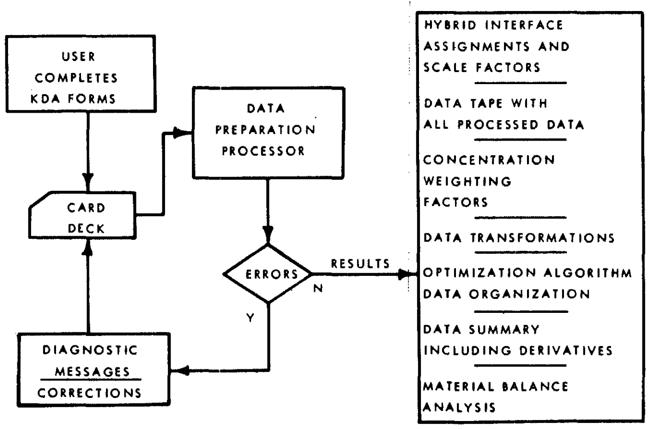

USER COMPLETES KDA FORMS
DATA PREPARATION PROCESSOR
CARD DECK
ERRORS
RESULTS
N
Y
DIAGNOSTIC MESSAGES CORRECTIONS
HYBRID INTERFACE ASSIGNMENTS AND SCALE FACTORS
DATA TAPE WITH ALL PROCESSED DATA
CONCENTRATION WEIGHTING FACTORS
DATA TRANSFORMATIONS
OPTIMIZATION ALGORITHM DATA ORGANIZATION
DATA SUMMARY INCLUDING DERIVATIVES
MATERIAL BALANCE ANALYSIS

Figure 5—Data preparation processor flowchart

## User interaction communication

The user's first contact with the Kinetic Data Analysis package is a set of data forms (see Figure 4) that request experimental data and other related information in kinetic rather than computer terminology. These forms are transformed into a deck of punched cards and fed to the Data Preparation Processor. Referring to Figure 5, if no errors are detected, the data is processed and the results are printed out and stored on tape. This tape contains all optimization algorithm and kinetic information required for the execution of the optimization program.

To complete the data forms the user is required to provide a "yes" or "no" answer to the question, "All-Digital Solution?" The initial answer to this question

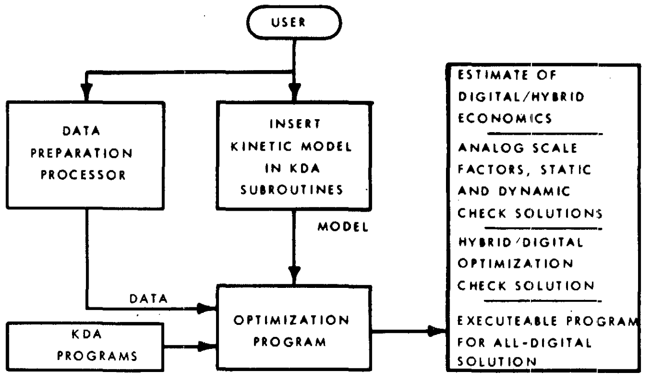

USER
DATA PREPARATION PROCESSOR
INSERT KINETIC MODEL IN KDA SUBROUTINES
MODEL
DATA
KDA PROGRAMS
OPTIMIZATION PROGRAM
ESTIMATE OF DIGITAL/HYBRID ECONOMICS
ANALOG SCALE FACTORS, STATIC AND DYNAMIC CHECK SOLUTIONS
HYBRID/DIGITAL OPTIMIZATION CHECK SOLUTION
EXECUTEABLE PROGRAM FOR ALL-DIGITAL SOLUTION

Figure 6—Flowchart for first phase of KDA study

is "yes" regardless of the user's intention to perform a hybrid simulation because, referring to Figure 6, the all-digital optimization program has a built-in mechanism for obtaining:

1. An analog static check and dynamic check solution.
2. A cost estimate of the all-digital solution versus the hybrid solution cost for problems where the most economic alternative is questionable.
3. An accurate estimate for all unknown analog scale factors.
4. An overall dynamic test for hybrid simulations

which are required to program and debug the analog model for hybrid studies.

For all-digital studies, the Kinetic Data Analysis package supplies three partially programmed FORTRAN IV subroutines and a "Block Data" subroutine for kinetic models consisting of either algebraic equations (e.g., stirred-tank reactor) or ordinary differential equations (e.g., batch or flow reactors). The integration package uses a fourth order Runge-Kutta integration algorithm and a readily implemented mechanism is available to obtain the classical "error versus step size" data to determine the correct and most economical step size for the integration process. The three subroutines require the user to:

1. Store initial values of the variables being integrated in an integration initial condition array.
2. Store computed results in a specified array.
3. Compute intermediate variables and model derivatives or, for example, stage outputs using FORTRAN IV statements.

Items one and two, typically, require two or three statements and the requirements for item three are a function of the complexity of the kinetic model.

The "Block Data" subroutine is used to define total number of and names of intermediate and integration variables for control and printout purposes.

These four programs (in object form) are incorporated into the Kinetic Data Analysis package to form an executable program which, upon request, will read in the data prepared by the Data Preparation Processor and print out the values of intermediate and dependent variables as a function of the independent variable. For all-digital studies, the user now has an executable optimization program capable of solving his problem.

For hybrid studies, this program provides static check, dynamic check and scale factor information. If the user executes one digital solution to his problem (this will be clarified later), the results provide the information required to test the overall accuracy of a hybrid simulation and the running time of the all-digital model to compare hybrid versus digital economics.

With the exception of reprocessing the card deck obtained from the data forms and requesting hybrid processing, no digital programming is required for hybrid studies. The Data Preparation Processor, in the hybrid mode, assigns hybrid interface channels to operate in conjunction with preprogrammed hybrid interface programs. Since data transferred to and from the analog model is done in a predefined sequence, the analog logic and interface circuits are also predefined and can be prepatched. Therefore, the additional effort required for hybrid studies is limited to the analog programming required to actually simulate the kinetic model.

The Kinetic Data Analysis package has, in effect, organized the hybrid study and, with the aid of the static check, dynamic check, and scale factors determined earlier, made programming and debugging the analog model a relatively simple task. The aforementioned card input analog set-up program limits the time required to set up and check out analog programs to a few minutes.

At execution time, the user communicates with the Data Preparation Processor, the optimization programs, and the Program Executive through a set of predefined user oriented commands. These commands can be inputed via cards for batch-unattended runs or a console typewriter. Since the Kinetic Data Analysis package uses a "space" as a delimiter, commands are entered in "free format." For example, the command "INPUT DATA 8", which is used to read in the data tape from FORTRAN I/O unit 8, may start at any location on a punch card.

The above mentioned command list, which contains more than fifty individual commands, is too extensive to discuss in detail. The commands can, however, be classified into the six areas of control they make available to the user.

| | | |
|---|---|---|
| 1. | Program Control.... | Select I/O devices, call Kinetic Data Analysis Programs, add to the program library, etc. |
| 2. | Kinetic Data Handling.............. | Control I/O options and computations performed on experimental and computed kinetic data. |
| 3. | Optimization Data Handling................. | Control I/O options and computations associated with optimization variables. |
| 4. | Objective Function Control.................. | Control the mathematical form, weighting and the components or data sets used to compute the objective function (see later discussion). |
| 5. | Optimization Algorithm Control..... | Select the mode (e.g., maximize, minimize) and other options (e.g., iterative, cyclic operation) associated with the optimization algorithm. |
| 6. | Model Control and Diagnostic............ | Select hybrid diagnostic options (e.g., scan for interface error messages) or digital model control options (e.g., set or reset a one/zero model switch to modify kinetic model). |

The form of the results obtained by the user during program execution will be discussed later.

## Data capacity and classification

The Kinetic Data Analysis package is capable of processing up to fifteen sets of experimental kinetic data (or data sets) which may contain concentration data for a maximum of fifteen chemical species or components. Each data set may contain up to ten values of an independent or sampling variable (e.g., time for batch reactor, volume for flow reactor, etc.) and fifteen concentration points per sampling variable. These data must be common to all data sets and the sampling variable must be a monoatonic increasing function whose initial value is zero. However, equal sampling variable increments are not required. Each data set also contains provision for a catalyst concentration, a temperature, and an alphanumeric user identifier. The purpose and manipulation of the catalyst and temperature data will be discussed later.

Up to fifteen unknown reaction rate constants, which are assumed to obey the Arrhenius equation, can be processed. This limit is independent of the thermal state of the system (i.e., isothermal or non-isothermal data sets). In addition, the Kinetic Data Analysis package can process up to fifteen unknown model or individual parameters (e.g., reaction orders, heat transfer coefficients, etc.).

The above mentioned limits apply to all-digital studies and hybrid systems whose interface contains a minimum of sixteen analog to digital and digital to analog channels.

The Data Preparation Processor catagorizes experimental kinetic data into one of three classes called KDA Case Numbers. They are:

Case #1.. One or more experiments performed under nonisothermal conditions

Case #2.. Two or more experiments performed under isothermal conditions where the difference between the maximum and minimum temperature levels is greater than 5°C or °F.

Case #3.. One or more experiments performed under isothermal conditions where the temperature range is less than or equal to 5°C or °F.

This data catagorization is one of the key factors required, for example, to organize optimization algorithm input data and the transfer of rate constants to the kinetic model.

## Optimization variable transformations

Two tranformations, which play an important part in the data flow between the various KDA processors, are:

1. The tranformation of rate constants and model parameters into optimization algorithm variables.
2. The transformation and transfer of these variables to the kinetic model.

Both transformations are a function of the aforementioned KDA Case Number and the Arrhenius equation

$$K = A \cdot \mathrm{EXP}\,(-B/T) \tag{1}$$

where

$$K = \text{reaction rate constant}$$
$$A, B = \text{Arrhenius coefficients}$$
$$T = \text{absolute temperature}$$

The Kinetic Data Analysis package uses an alternative, but rigorously correct, form of the Arrhenius equation whose derivation is shown in Appendix A. This relationship is

$$K = (K_R)\,(K_{HL})^{\beta} \tag{2}$$

where $\beta$ is defined as

$$\beta = (1/T_R - 1/T)/(1/T_L - 1/T_H) \tag{3}$$

In equation 3, $T_H$ and $T_L$ are the maximum and minimum experimental data temperatures, respectively and $T_R$ is a mid-range reference temperature defined by the equation

$$1/T_R = (1/T_L + 1/T_H)/2 \tag{4}$$

In equation 2, $K_R$ denotes the reaction rate constant at $T_R$ and $K_{HL}$ is the ratio of the maximum to minimum rate constants ($K_{HL} = K_H/K_L$).

For experimental data catagorized as KDA Case #3, the optimization variables, $\lambda_i$, are defined as:

$$\lambda_i = K_i \tag{5}$$

where

$$i = \text{rate constant index, } i = 1, 2, \cdots, \text{NRC}$$
$$\text{NRC} = \text{the total number of rate constants.}$$

For the two remaining data catagories

$$\lambda_{2i-1} = K_R^i \tag{6}$$

and

$$\lambda_{2i} = K^{i}_{HL} \tag{7}$$

Individual or model parameters specified by the user are sequentially added after the last rate constant variable. For example, the first parameter, $P_1$, is assigned to $\lambda_{NRC+1}$ for KDA Case #3.

Referring to Figure 7, optimization variables are transferred to the kinetic model as a function of the KDA Case Number as shown in Table I. For hybrid kinetic models, the rate constants are scaled and transferred to the analog computer in a predefined transfer sequence as shown in Table II. Note that for both digital and hybrid models concentration initial conditions, sampling points, and a ramp sloape (i.e., reciprocal of the last data set sampling point) are also transferred to the kinetic model.

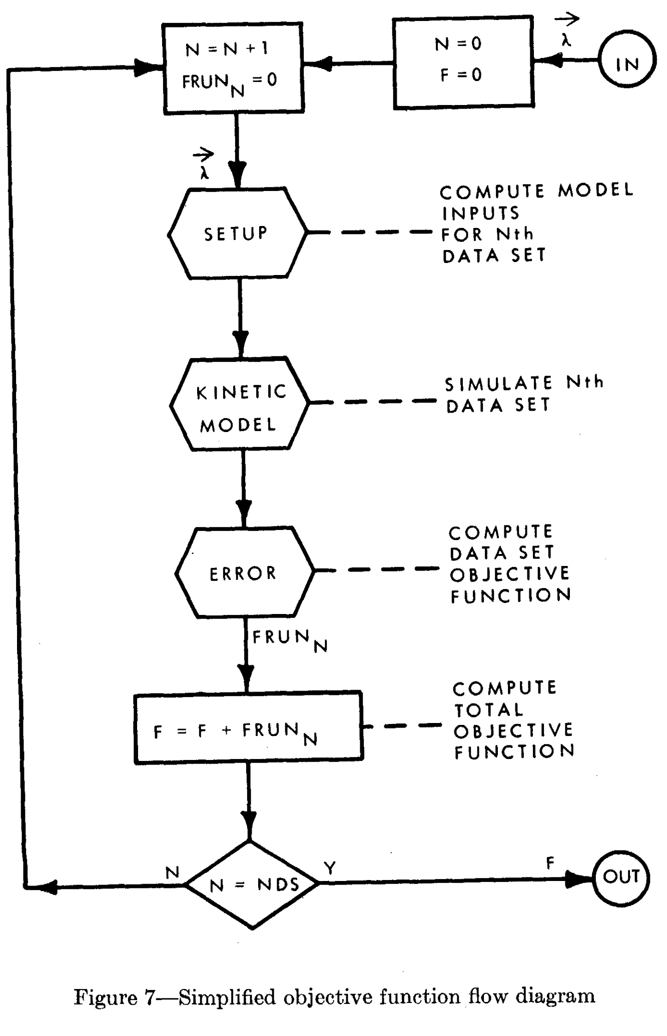


Figure 7—Simplified objective function flow diagram

TABLE I— Items influenced by KDA case number

| | KDA CASE NUMBER | | |
|---|---|---|---|
| | 1 | 2 | 3 |
| TOTAL NUMBER OF OPTIMIZATION VARIABLES | 2 • NRC + NPR | 2 • NRC + NPR | NRC + NPR |
| FORM ASSIGNED TO OPTIMIZATION VARIABLES REPRESENTING RATE CONSTANTS | $K_R$ $K_{HL}$ | $K_R$ $K_{HL}$ | K |
| RATE CONSTANT DATA TRANSFERRED TO DIGITAL MODEL | A, B $K_R$, $K_{HL}$ | A, B K | K |
| RATE CONSTANT DATA TRANSFERRED TO HYBRID MODEL | LOG ($K_{HL}$) LOG ($K_R$) | K | K |

TABLE II—Typical transfer sequence* for KDA case #2

| CHANNEL NUMBER | D/A DEMULTIPLEXING | | | A/D |
|---|---|---|---|---|
| | A | B | C | |
| 1 | $C\emptyset_1$ | | $K_1$ | $C_1$ |
| 2 | $C\emptyset_2$ | | $K_2$ | $C_2$ |
| ⋮ | ↓ | DATA SET TEMPERATURE AND CATALYST CONCENTRATION | ↓ | ↓ |
| NCS | | | | |
| ⋮ | INITIAL CONCENTRATION | | | COMPUTED CONCENTRATIONS CORRESPONDING TO SAMPLING VARIABLE |
| NRC | | | | |
| ⋮ | | | REACTION RATE CONSTANTS | |
| 14 | | CAT** | | |
| 15 | | TEMP | | |

* Channel zero used by prepatched KDA circuits.

** Transferred when applicable.

## Optimization algorithm and objective function options

The current version of the Kinetic Data Analysis package uses a slightly modified version of the PARTAN algorithm described in detail by Harkins[4]. Since a detailed description of the algorithm is available, this paper will only consider the mathematical form of the objective function. However, it should be noted that this algorithm, which can be classified as an "accelerated gradient" algorithm, was selected because of its proven effectiveness on a number of all-digital and hybrid kinetic studies performed in recent years at EAI Computation Centers. The add-on capability of the software package makes it possible to add other algorithms if the need exists.

The mathematical form of the objective function is specified by the user at execution time. Referring to Figure 7, the objective function is based on the "total error" or sum of the individual data set errors. For example, to compute the objective function for a problem consisting of ten components and ten data sets, ten analog runs or one hundred digital integrations are required.

The form of the objective function, its weighting factors, the exclusion of a chemical species or data sets from the objective function, etc., are defined by the user at execution time via the Executive Program. The software package provides integral and polynomial objective function options to the user based on the following definitions:

| | | |
|---|---|---|
| $E_{n,m,i}$ | = | $COMP_i \cdot \lvert C_{n,m,i} - C^*_{n,m,i} \rvert^{EXPN}$ |
| $COMP_i$ | = | 1.0 or 0 when a chemical species is to be excluded |
| $OMIT_n$ | = | 1.0 or 0 when a data set is to be excluded |
| i | = | index denoting a chemical species, $1 \leq i \leq J$ |
| m | = | index denoting a sampling point, $1 \leq m \leq M$ |
| n | = | index denoting a data set or experiment, $1 \leq n \leq N$ |
| $C_{n,m,i}$ | = | computed results (unscaled) array |
| $C^*_{n,m,i}$ | = | experimental results array |
| F | = | total objective function |
| $FRUN_n$ | = | data set objective functions |
| EXPN | = | a positive, non-zero constant |

The polynomial option defines individual data set objective function as

$$FRUN_n = \sum_{m=2}^{M} WGT_{n,m} \sum_{i=1}^{J} E_{n,m,i} \tag{8}$$

where the weighting factor ($WGT_{n,m}$) is

$$WGT_{n,m} = 1 + \frac{PW1}{\theta_s^m} + PW2 \cdot \theta_s^m \tag{9}$$

In the above relationship $\theta$ denotes a positive sampling variable ratio whose maximum value is unity:

$$\theta_s^m = \frac{SV_m}{SV_M} \tag{10}$$

The weighting factor is unity if PW1 and PW2 are zero. If PW1 = 1.0 and PW2 = 0, initial values are weighted, and if PW2 = 1.0 and PW1 = 0, final values are weighted. Note that both PW1 and PW2 cannot simultaneously be set to one.

The integral option defines individual data set objective functions as

$$FRUN_n = \sum_{i=1}^{J} WGT_{n,i} \int_0^{SV_M} E_{n,m,i}\, d(SV) \tag{11}$$

where the integral is computed using a "Trapezoidal Rule" approximation and the weighting factor ($WGT_{n,i}$) is defined as

$$WGT_{n,i} = 1 + CW1 \cdot \overline{C}_{n,i} + CW2 \cdot (1 - \overline{C}_{n,i}) \tag{12}$$

The control constants CW1 and CW2 are identical in behavior to PW1 and PW2. the $\overline{C}_{n,1}$ values are concentration weighting factors computed from experimental data by the Data Preparation Processor.

If CW1 = 1.0 and CW2 = 0, large concentrations are weighted, and if CW2 = 1.0 and CW1 = 0, small concentrations are weighted. This weighting factor is useful when, for example, a component whose range is 0 − 0.05 in a given experiment is more sensitive to an analytical error of, say, ± 0.01 than a component whose range is 0.5 − 1.0.

Referring to Figure 7, the total objective function F, is obtained by summing the individual data set errors, FRUN, modified by $OMIT_n$ (1.0 or 0) to control the inclusion or exclusion of the various data sets.

$$F = \sum_{n=1}^{N} OMIT_n \cdot FRUN_N \tag{13}$$

Note that user commands control the values assigned to $OMIT_n$, $COMP_i$, EXPN, PW1, PW2, CW1 and CW2.

In addition to the aforementioned objective functions, the software package has provision for the user to add a digital subroutine to compute the individual data set errors if the "built-in" options are not applicable. For example, if the data set errors are computed on the analog computer this subroutine can be used to transfer them into the digital computer.

Optimization results include a table containing the objective function, its fractional contribution to the total objective function, and the average error per data point for each data set. The total absolute error* or standard error is included in all results to allow the user to compare the relative merits of various objectives functions since their magnitudes depend on their mathematical form.

## Temperature and catalyst data

Each of the data sets has associated with it a single temperature which is sufficient for experiments performed under isothermal conditions (i.e., KDA Case #2 and 3.) For non-isothermal situations the data set temperature is the initial or feed temperature; therefore, the requirements of kinetic models which include energy balances (i.e., temperature obtained from the solution of a differential equation) are also satisfied.

Studies that require the storage of, say, temperature versus time data are simulated by:

1. Using "Data" statements to include these data in the subroutines supplied by the user for all digital studies.
2. Using, say, card programmed diode function generators (CPDFG) on the analog computer for hybrid studies.

The CPDFGs work in conjunction with preprogrammed logic that automatically associates each function with the appropriate data set during the simulation.

The software package also allows the user to associate a catalyst concentration with each data set. The catalyst concentration, which is transferred to the kinetic model, provides the user with a mechanism for simulating kinetic models involving a non-reactive or reactive catalyst. For example, when catalyst concentration data is not available in studies involving reactive catalysts, the catalyst concentration is the initial condition for the catalyst material balance equation.

### *Typical application*

The following discussion will be devoted to the solution of "Monsanto Benchmark Problem" using the Kinetic Data Anaylsis package on a fu ly expanded EAI 8900 Hybrid Computer. This discussion will include a mathematical description of the problem, illustrate the form of the results obtained during the preparation and optimization phases of the study, and summarize the numerical results obtained from the study. Simulation accuracy, errors in results, and economics will also be discussed.

* Equation 8 with EXPN and $WGT_{n,m}$ equal to unity.

## Problem description

The illustrative problem contains the two essentia. ingredients to perform a kinetic data analysis study; a proposed kinetic model and experimental data. Referring to Table III, each of the thirteen available data sets contained concentration-time data for seven chemical species (i.e., R. S, T, U, W, X, and Y), the concentration of a non-reactive catalyst, and a temperature. These data were obtained from experiments performed under isothermal conditions over a 133 to 181°C temperature range which included a threefold variation in catalyst concentration, 117 to 368. No two data sets had identical initial concentrations and the number of non-zero sampling variable (i.e., time) points per data set varied from one to four.

The proposed kinetic model, which is shown in Table IV, is based on the following chemical equations:

$$R + S \underset{K_8}{\overset{K_1}{\rightleftarrows}} T \xrightarrow{K_4} U$$

$$R + S \xrightarrow{K_2} U$$

$$R + S \underset{K_9}{\overset{K_3}{\rightleftarrows}} W$$

$$T + S \xrightarrow{K_5} X \underset{K_{10}}{\overset{K_6}{\rightleftarrows}} U + S$$

$$U + S \underset{K_{11}}{\overset{K_7}{\rightleftarrows}} Y$$

The model contained eleven unknown rate constants ($K_1 - K_{11}$) and since this study falls under the KDA Case #2 category, there are a total of twenty-two optimization variables. Each rate constant has one $K_R$ and one $K_{HL}$ optimization variable associated with it.

## Data preparation processor results

Processing the card deck corresponding to the KDA Data Forms produced the results indicated in Figure 5, which are illustrated by Figures 8 through 11. These

TABLE III— Typical data set

IDENTIFIER: RUN TWO

TEMPERATURE: 146°C CATALYST CONCENTRATION: 117

| TIME | CONCENTRATION IN MASS FRACTION | | | | | | |
|---|---|---|---|---|---|---|---|
| HOURS | R | S | T | U | W | X | Y |
| 0.0 | 0.425 | 0.501 | 0.018 | 0.005 | 0.050 | -- | -- |
| 1.0 | 0.359 | 0.465 | 0.051 | 0.017 | 0.106 | -- | 0.002 |
| 2.0 | 0.315 | 0.442 | 0.086 | 0.033 | 0.120 | -- | 0.004 |
| 3.0 | 0.281 | 0.424 | 0.123 | 0.048 | 0.116 | -- | 0.008 |

```
ADC ASSIGNMENTS
-----------------------------------------------------------
ADC CHANNEL      VARIABLE        MAXIMUM          SCALE
  NUMBER           NAME           VALUE           FACTOR
-----------------------------------------------------------

    1             COMP R        0.1000E 01      0.1000E 01
    2             COMP S        0.1000E 01      0.1000E 01
    3             COMP T        0.1000E 01      0.1000E 01
    4             COMP U        0.1000F 01      0.1000E 01
    5             COMP W        0.5000E 00      0.2000E 01
    6             COMP X        0.5000E 00      0.2000E 01
    7             COMP Y        0.1000E 00      0.1000E 02
-----------------------------------------------------------
```

```
DAC ASSIGNMENTS
--------------------------------------------------------------------------
DAC CHANNEL     VARIABLE        MAXIMUM         SCALE      LOG (KHLMX)
  NUMBER          NAME           VALUE          FACTOR    (CASE 1 ONLY)
--------------------------------------------------------------------------

    1          RATE CON  1    0.1500E 02     0.6666E-01
    2          RATE CON  2    0.8000E 01     0.1250E 00
    3          RATE CON  3    0.8000E 01     0.1250E 00
    4          RATE CON  4    0.4000F 00     0.2500E 01
    5          RATE CON  5    0.1000E 01     0.1000E 01
    6          RATE CON  6    0.1000E 01     0.1000E 01
    7          RATE CON  7    0.2500E 00     0.4000F 01
    8          RATE CON  8    0.8000E 01     0.1250E 00
    9          RATE CON  9    0.5000E 02     0.2000E-01
   10          RATE CON 10    0.1000E 01     0.1000E 01
   11          RATE CON 11    0.4000E 01     0.2500E 00
--------------------------------------------------------------------------
```

Figure 8—Hybrid interface assignments

figures omit the first phase of the form processing output. That is, the direct playback of the KDA Data Forms with appropriate error messages when errors are detected.

```
TEMPERATURE IN DEG C          MINIMUM   130.0      MAXIMUM   200.0

CATALYST ( UNKNOWN  ) IN UNKNOWN          MINIMUM 0.117E 03


SCALED CATALYST-TEMPERATURE DATA
-------------------------------------------------------------
  DATA SET                  SCALED           SCALED
NO. IDENTIFIER           TEMPERATURE      CATALYST CONC
-------------------------------------------------------------

 1    RUN  ONE              0.7300           0.3315
 2    RUN  TWO              0.7300           0.3179
 3    RUN  3                0.9000           0.6440
 4    RUN  FOUR             0.8100           0.3288
 5    RUN  FIVE             0.8100           0.6576
 6    RUN  SIX              0.7900           0.6522
 7    RUN  7                0.7400           0.6522
 8    RUN  8                0.8350           0.6522
 9    RUN  NINE             0.8500           0.6522
10    RUN  TEN              0.8900           0.6522
11    RUN  11               0.6650           0.6739
12    RUN  12               0.9050           0.6141
13    RUN  13               0.8650           1.0000

-------------------------------------------------------------

CAT CONC AND TEMP XFER ON DAC 14 AND 15 DURING 'B' PERIOD
```

Figure 9—Temperature-catalyst interface data transfer

Figure 8 illustrates the hybrid interface assignments for the eleven reaction rate constants and the seven chemical species involved in the mathematical model, their maximum values, and their scale factors (i.e., reciprocal of maximum value). Figure 9 details the scaled temperatures and catalyst concentrations that will be transferred to the analog model during the "B" demultiplexing period on D/A-channels 14 and 15. Note that this problem is in the KDA Case #2 category whose interface transfer sequence has been illustrated in Table II.

Referring to Figure 10, the Data Preparation Processor assigns a number to both the data sets and chemical species involved in the study. These numbers are required by the user to execute commands that manipulate specific chemical components or data sets. For example, to exclude the eleventh data set from the study, the command is "EXCLUDE 11" not "EXCLUDE RUN 11" where "RUN 11" is the data set identifier specified by the user.

The lower half of Figure 10 illustrates a typical data set printout containing the original "time" units and scaled values (i.e., normalized) of the sampling variable. The normalized values were obtained by:

```
NAME / KDA NUMBER SUMMARY
-------------------------

 ----------------------------
 KDA NUMBER         COMP NAME
 ----------------------------
     1              COMP R
     2              COMP S
     3              COMP T
     4              COMP U
     5              COMP W
     6              COMP X
     7              COMP Y
 ----------------------------

 -----------------------------
 KDA NUMBER     DATA SET IDENT
 -----------------------------
     1              RUN  ONE
     2              RUN  TWO
     3              RUN  3
     4              RUN FOUR
     5              RUN FIVE
     6              RUN  SIX
     7              RUN  7
     8              RUN  8
     9              RUN NINE
    10              RUN  TEN
    11              RUN  11
    12              RUN  12
    13              RUN  13
 -----------------------------
```

```
DATA SET NUMBER  2          USER IDENTIFIER   RUN  TWO          TEMPERATURE   146.0 C

CATALYST CONCENTRATION  0.1170E 03 UNKNOWN          CATALYST RATIO  0.1000E 01

BETA FACTOR   -0.2420

SAMPLING POINTS
-------------------------------------------------------------------------------
  TIME         0.000E 00 0.100E 01 0.200E 01 0.300E 01
NORMALIZED      0.0000    0.3333    0.6667    0.9999
-------------------------------------------------------------------------------
COMPONENT                                    CONCENTRATION IN WGT FRAC
-------------------------------------------------------------------------------

COMP R         0.425E 00 0.359E 00 0.315E 00 0.281E 00

COMP S         0.501E 00 0.465E 00 0.442E 00 0.424E 00

COMP T         0.100E-01 0.510E-01 0.860E-01 0.123E 00

COMP U         0.500E-02 0.170E-01 0.330E-01 0.480E-01

COMP W         0.500E-01 0.106E 00 0.120E 00 0.116E 00

COMP X         0.000E 00 0.000E 00 0.000E 00 0.000E 00

COMP Y         0.000E 00 0.200E-02 0.400E-02 0.800E-02

-------------------------------------------------------------------------------
CONC SUM       0.999E 00 0 100E 01 0.100E 01 0.100E 01
```

Figure 10—KDA number assignments and processed data set

TABLE IV—Mathematical model

DEFINITION OF TERMS

| | | |
|---|---|---|
| $R_1 = K_1 RS$ | $R_6 = K_6 US$ | $R_{11} = K_{11} Y$ |
| $R_2 = K_2 RS$ | $R_7 = K_7 US$ | $R_{12} = R_1 + R_2 + R_3$ |
| $R_3 = K_3 RS$ | $R_8 = K_8 T$ | $R_{13} = R_4 + R_8$ |
| $R_4 = K_4 T$ | $R_9 = K_9 W$ | $R_{14} = R_6 + R_7$ |
| $R_5 = K_5 TS$ | $R_{10} = K_{10} X$ | |
| $t$ = time | CAT = Catalyst Concentration | |

$\alpha = \mathrm{CAT}/(\mathrm{CAT})_{\mathrm{MIN}}$ = Catalyst Ratio

MRT, MSR, MST, etc. = Molecular Weight Ratios; $\theta = \alpha t$

MATERIAL BALANCE EQUATIONS

$$\frac{dR}{d\theta} = \frac{dR}{\alpha dt} = -R_{12} + (MRT) R_8 + (MRW) R_9$$

$$\frac{dS}{d\theta} = \frac{dS}{\alpha dt} = -(MSR) R_{12} - R_{14} - R_5 + (MST) R_8 + (MSW) R_9 + (MSX) R_{10} + (MSY) R_{11}$$

$$\frac{dT}{d\theta} = \frac{dT}{\alpha dt} = (MTR) R_1 - R_{13} - (MTS) R_5 ; \qquad \frac{dY}{d\theta} = \frac{dY}{\alpha dt} = (MYS) R_7 - R_{11}$$

$$\frac{dU}{d\theta} = \frac{dU}{\alpha dt} = (MUR) R_2 + (MUT) R_4 + (MUX) R_{10} + (MUY) R_{11} - (MUS) R_{14}$$

$$\frac{dX}{d\theta} = \frac{dX}{\alpha dt} = (MXS) (R_5 + R_6) - R_{10} ; \qquad \frac{dW}{d\theta} = \frac{dW}{\alpha dt} - (MWR) R_3 - R_9$$

```
----------------------------------------------------------------------
  DATA SET                        CHEMICAL SPECIES
NO. IDENTIFIER                   1-5 / 6-10 / 11-15
----------------------------------------------------------------------

 1   RUN  ONE        0.5215  0.2410  0.0830  0.0335  0.1065
                     0.0000  0.0150

 2   RUN  TWO        0.3450  0.4580  0.0695  0.0250  0.0980
                     0.0000  0.0035

 3   RUN  3          0.1550  0.3100  0.1023  0.2400  0.0197
                     0.1553  0.0100
```

```
PARTAN DATA SUMMARY
-------------------

 CONTROL DATA.... TYPE = MIN     NPOI = 1     NEXP = 3     LEVAL =  2500
 PARTAN COEF. ... BMAX =  0.1000     BMIN =  0.0010     ALMAX =  0.1000     ALMIN =  0.0010

 PARAMETER DATA
 ----------------------------------------------------------------------------
 NO.   NAME     TYPE      RANGE          VALUE          MAXVAL         MINVAL
 ----------------------------------------------------------------------------
  1    KR01      3     0.1000E 01     0.4300E 00     0.1000E 01     0.0000E 00
  2    HL01      3     0.1990E 03     0.1040E 03     0.2000E 03     0.1000E 01
  3    KR02      3     0.4000E 00     0.1400E 00     0.4000E 00     0.0000E 00
  4    HL02      3     0.3990E 03     0.1080E 03     0.4000E 03     0.1000E 01
  5    KR03      3     0.1000E 01     0.7100E 00     0.1000E 01     0.0000E 00
```

Figure 11—Concentration weighting factors and algorithm input data

1. Performing the catalyst transformation shown in Table IV, which was the result of a "yes" answer to the question, "CATALYITC REACTIONS?" (see Figure 4).
2. Dividing all values by the maximum sampling point to form the "normalized" values or scaled sampling points.

These results also contain concentration and rate summations for each time point to assist the user in evaluating the consistency of the data based on material balance. The rates, which are not shown in Figure 10, were computed numerically by differentiating a polynomial whose coefficients are determined by a least square fit of the concentration data.

Figure 11 illustrates the concentration weighting factors and the input data to the PARTAN Algorithm. Note that the Data Preparation Processor has assigned names, for example, "KR01", to the optimization variable and placed them in a "type three" category. This means they are constrained between an upper and lower limit denoted by "MAXVAL" and "MINVAL". The initial values of the variables are in the "VALUE" column.

The results of the preprocessing indicated that the eleventh data set should be excluded from the study because its concentration sums indicated as much as ten percent error. Therefore, optimization results were obtained using twelve, rather than thirteen, data sets.

## Optimization results

Figures 12 through 15 illustrate the form of some of the results obtained from the hybrid solution of the problem. Figure 12 illustrates the user commands, which are documented as they are processed, and an optimization summary. The summary is updated everytime the algorithm detects an improvement in the objective function during the execution of an optimization run. It keeps a running record of the total number of gradient evaluations and objective function evaluations and notes any optimization variables at their upper or lower limit (not shown in Figure 12). The

```
TYPE
INPUT DATA 8

     KDA DATA TAPE IDENTIFIER
     ------------------------

        USER IDENTIFIER
        ---------------
            COMPANY........     MONSANTO COMPANY
            LOCATION.......     ST. LOUIS, MO.
            PROJ ENGR......     PAUL PARISOT

        EAI IDENTIFIER
        --------------
            PROJ NUMBER....     100609
            PROJ ENGR......     A. CARLSON
            CURRENT DATE...     SEPT., 1968

TAPE UNIT 9
RESTORE PARTAN PLOT
EXCLUDE 11
INTEGRAL OBJECTIVE FUNCTION
WEIGHT LARGE CONCENTRATIONS
ERROR EXPONENT 1.0


OPTIMIZATION SUMMARY
--------------------

IMPROVEMENT    OBJECTIVE    NO OF FUNC.    NO OF GRAD.    CURRENT    CURRENT
  NUMBER       FUNCTION     EVALUATIONS    EVALUATIONS     ALPHA      BETA
-----------------------------------------------------------------------------

    0         2.8090E 01          1              0         0.1000     0.1000
    1         0.6715E 01         24              1         0.1000     0.1000
    2         0.5308E 01         25              1         0.1000     0.1618
    3         0.5221E 01         26              1         0.1000     0.2618
    4         0.5106E 01         29              1         0.1000     0.4236
    5         0.5100E 01         30              1         0.1000     0.4236
    6         0.5071E 01         54              2         0.1000     0.1618
    7         0.4817E 01         56              2         0.1000     0.1618
    8         0.4814E 01         57              2         0.1000     0.1618
    9         0.4769E 01         59              2         0.1000     0.1618
   10         0.4766E 01         62              2         0.1000     0.1618
   11         0.4711E 01         85              3         0.1000     0.1618
   12         0.4694E 01         87              3         0.1000     0.2618
   13         0.4667E 01         90              3         0.1000     0.2618
   14         0.4498E 01        118              4         0.1000     0.1000
   15         0.4479E 01        124              4         0.1000     0.1000
   16         0.4424E 01        151              5         0.1000     0.0382
   17         0.4415E 01        154              5         0.1000     0.0382
   18         0.4380E 01        155              5         0.1000     0.0382
   19         0.4235E 01        182              6         0.0382     0.0382
   20         0.4137E 01        183              6         0.0382     0.0618
   21         0.4111E 01        188              6         0.0382     0.1000
   22         0.4111E 01        191              6         0.0382     0.1000
   23         0.4104E 01        192              6         0.0382     0.1000
   24         0.4093E 01        216              7         0.1000     0.0382
   25         0.4069E 01        272              9         0.1000     0.0382
   26         0.3963E 01        298             10         0.0382     0.0382
   27         0.3918E 01        299             10         0.0382     0.0618
   28         0.3839E 01        326             11         0.1000     0.1000
   29         0.3832E 01        329             11         0.1000     0.1618
   30         0.3827E 01        330             11         0.1000     0.1618

-----------------------------------------------------------------------------
```

Figure 12—Typical executive program output and optimization summary

```
     DATA SET                        ABSOLUTE      ERROR        AVERAGE
NUMBER    IDENTIFIER                  ERROR       FRACTION       ERROR
------------------------------------------------------------------------

   1       RUN  ONE                0.7910E-01   0.2053E-01   0.1130E-01
   2       RUN  TWO                0.2296E 00   0.5961E-01   0.1093E-01
   3       RUN  3                  0.4674E 00   0.1213E 00   0.3339E-01
   4       RUN FOUR                0.4895E 00   0.1270E 00   0.2331E-01
   5       RUN FIVE                0.4052E 00   0.1051E 00   0.1929E-01
   6       RUN  SIX                0.1987E 00   0.5158E-01   0.9463E-02
   7       RUN  7                  0.4990E 00   0.1295E 00   0.2376E-01
   8       RUN  8                  0.5330E 00   0.1383E 00   0.1903E-01
   9       RUN NINE                0.1603E 00   0.4162E-01   0.5727E-02
  10       RUN  TEN                0.1424E 00   0.3697E-01   0.6783E-02
  12       RUN  12                 0.3960E 00   0.1027E 00   0.1414E-01
  13       RUN  13                 0.2517E 00   0.6535E-01   0.8092E-02

------------------------------------------------------------------------

PERCENT REPRO ERROR   0.650


DATA SET NUMBER  2         USER IDENTIFIER   RUN  TWO         TEMPERATURE   146.0 C

CATALYST CONCENTRATION  0.1170E 03 UNKNOWN          CATALYST RATIO  0.1000E 01

BETA FACTOR   -0.2420

SAMPLING POINTS
------------------------------------------------------------------------------------
  TIME        0.000E 00 0.100E 01 0.200E 01 0.300E 01
------------------------------------------------------------------------------------
COMPONENT                                        CONCENTRATION IN WGT FRAC
------------------------------------------------------------------------------------

COMP R        0.425E 00 0.359E 00 0.315E 00 0.281E 00
             -----------------------------------------------------------------------
              0.425E 00 0.345E 00 0.302E 00 0.279E 00
             -----------------------------------------------------------------------
              0.000E 00-0.137E-01-0.121E-01-0.170E-02
------------------------------------------------------------------------------------
COMP S        0.501E 00 0.465E 00 0.442E 00 0.424E 00
             -----------------------------------------------------------------------
              0.501E 00 0.455E 00 0.431E 00 0.416E 00
             -----------------------------------------------------------------------
              0.000E 00-0.906E-02-0.100E-01-0.798E-02
```

Figure 13—Typical objective function summary and detailed data set results

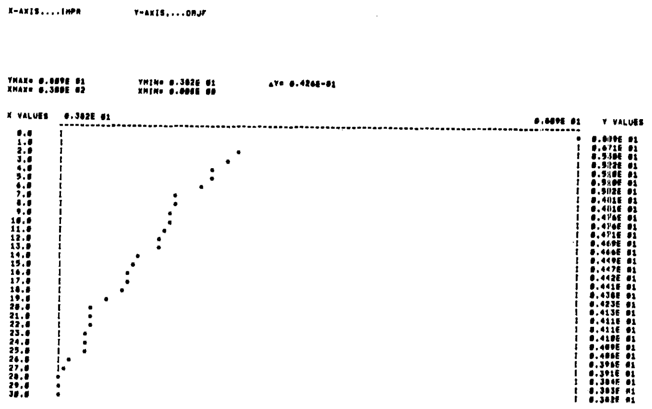
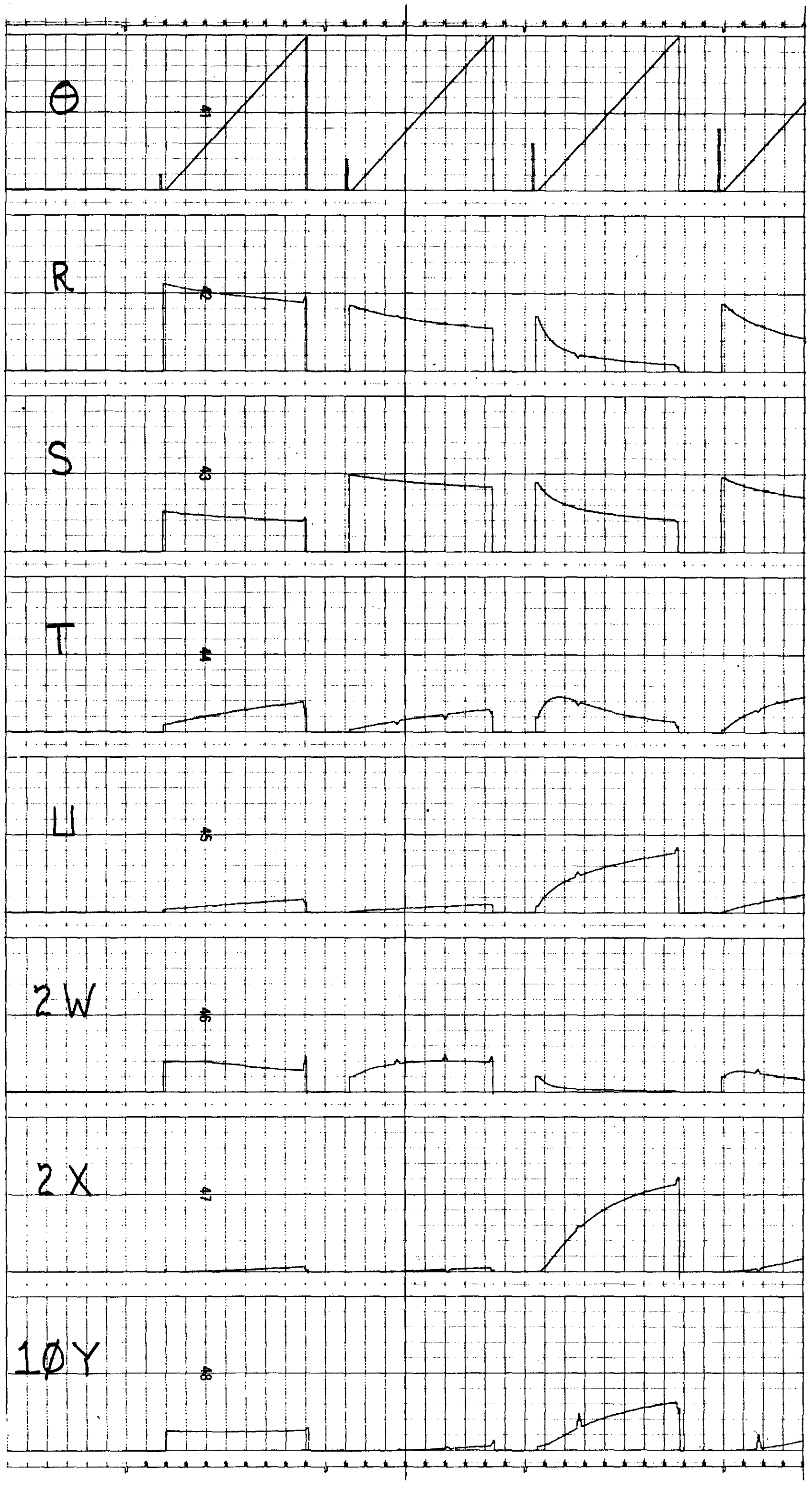


Figure 14—Typical line printer objective function—No. of improvements plot

Figure 15—Typical concentration results output



alpha and beta values pertain to the algorithm perturbations, etc.

During the optimization process the optimization summary is the only output available to the user with the exception of a percent improvement indicated on the analog computer digital voltmeter. The percent improvement is relative to the initial or base value of the objective function.

After the optimization process has been completed, the previously mentioned objective function summary is obtained (see Figure 13) which includes a reproducibility error. Referring to Figures 13 and 14, the user may also request a detailed comparison of experimental to computed results and a line printer plot of the objective function or any of the optimization variables as a function of the number of improvements.

The objective function summary allows the user to determine if, for example, any one data set is making an excessively large contribution to the objective function. The reproducilibity factor, which is typically zero for all-digital studies, is obtained by re-evaluating the objective function under "best fit" conditions after the optimization process has been completed. The percent error between the two objective functions is the percent reproducibility error shown in Figure 13. It reflects the total error introduced into the objective function by the hybrid interface, analog components, etc. As shown in Figure 13, this error was typically less than one percent.

The objective function plots allow the user to graphically follow the path of the optimization process. However, plots of specific optimization variables versus the number of improvements are more important. They indicate the activity or sensitivity of variables during optimization and allow the user to take appropriate action if, for example, a variable always remained essentially constant.

Figure 15 shows a concise final results plot that can be requested via the appropriate user command. This plot, which is obtained on the analog strip chart recorder, consists of a sample variable ramp ( $\theta$ ) and a set of curves for the computed concentrations. The "blips" on the concentration curves represent the deviation between the curves and experimental data points; therefore, the absence of "blips" represents a near perfect or perfect fit. The pulse prior to each ramp denotes the data set number. The first data set is preceded by a 10 volt pulse, the second by a 20 volt pulse, etc.

## Problem solution and results

To avoid the possiblity of confusing a local minimum on the error surface with the true minimum, sets of optimization runs were always made starting from four points on the error surface. The four sets of starting values used were the maximum and minimum values of the optimization variables, their arithmetic average values, and the initial or "best guess" values. The problem was solved using the following iterative process:

1. Perform four separate, complete optimization runs using the maximum, minimum, average, and initial values of the optimization variables.
2. Examine the results and determine if the final values of the objective function and optimization variables show good agreement.
3. If the results of step two indicate more runs are required, refine the four sets of starting values based on their results and repeat the first step.

This iteration process was repeated three times using the integral form of the objective function with large concentration weighting and an error exponent equal to unity. Referring to Table V, the values of the objective function for these three iterations are reported in standard error form (i.e., the unweighted sum of the absolute concentration errors). After the third iteration, the mathematical form of the objective function was changed to the standard form to eliminate the effects of the concentration weighting and the results of this iteration indicated that for all practical purposes, the "best fit" had been obtained.

The four sets of optimization variables obtained from the fourth iteration showed reasonably good but not perfect agreement. The error introduced into specific reaction rate constants by differences in the final values of the optimization variables were computed using the error form of equation 2; namely,

$$\frac{\Delta K}{K} = \frac{\Delta K_R}{K_R} + \beta\left(\frac{\Delta K_{HL}}{K_{HL}}\right) \tag{14}$$

where $\Delta K_R$ and $\Delta K_{HL}$ are the most probable errors and $K_R$ and $K_{HL}$ are the average values of the individual optimization variables.[9] The results of this analysis are shown in Table VI. Note that the absolute percent error of any one rate constant is a function of temperature or $\beta$ whose range is $\pm 0.5$.

### *Simulation Accuracy*

Comparisons between equivalent hybrid and all-digital optimization runs were made to determine how analog component or digital integration errors in-

TABLE V— Objective function results

| | STARTING LOCATION | | | |
|---|---|---|---|---|
| | INITIAL | MAXIMUM | MINIMUM | AVERAGE |
| STARTING VALUES* | 8.28 | 5.76 | 25.2 | 5.65 |
| ITERATION 1* | 3.76 | 3.15 | 3.59 | 3.65 |
| ITERATION 2* | 3.15 | 3.08 | 3.12 | 3.08 |
| ITERATION 3* | 3.06 | 3.03 | 3.11 | 3.08 |
| ITERATION 4* | 3.00 | 2.98 | 2.99 | 3.01 |

*Standard error equivalent of weighted integral objective function.

TABLE VI—Rate constant error analysis results

| | ABSOLUTE PERCENT ERROR | | |
|---|---|---|---|
| i | $K_R^i$ MINIMUM ERROR | $K_{HL}^i$ | $K_i$ MAXIMUM ERROR |
| 1 | 0.32 | 3.48 | 2.06 |
| 2 | 0.44 | 0.83 | 0.86 |
| 3 | 0.26 | 5.43 | 2.98 |
| 4 | 0.30 | 4.27 | 2.43 |
| 5 | 1.06 | 1.96 | 2.04 |
| 6 | 1.60 | 4.48 | 3.84 |
| 7 | 2.05 | 1.98 | 3.04 |
| 8 | 0.59 | 3.18 | 2.18 |
| 9 | 0.23 | 3.49 | 1.97 |
| 10 | 3.22 | 5.14 | 5.79 |
| 11 | 2.46 | 4.33 | 4.62 |

fluenced results. This comparison was based on the standard objective function value obtained after one function evaluation. Using both single and double precision digital integration, a comparison of objective function values showed good agreement between the digital and hybrid results. Both the hybrid and single precision digital integration results were within approximately $\pm$ 1% of the results obtained using double precision integration. These minor differences were traced to errors of less than 0.001 in computed concentration data points.

One comparison of equivalent all-digital versus hybrid optimization runs was made. Although both solutions differed slightly when their optimization

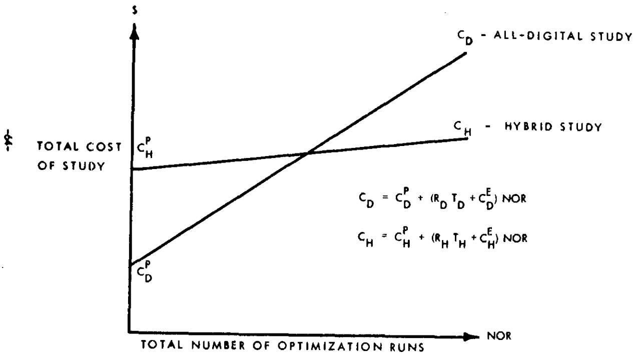


Figure 16—Typical hybrid-digital economic plot

summaries (see Figure 12) were compared, the final objective function and optimization variable results obtained were identical for all practical purpose (i.e., one or two percent difference). This would seem to indicate that the errors associated with experimental data and the mathematical model will have a greater influence on results than the relatively minor errors introduced by digital integration or analog components. It was also concluded that double precision integration accuracy was not worth the additional computation time it required compared to single precision integration.

### *Simulation economics*

The above discussion indicates there is no technical advantage to be gained by using a hybrid rather than an all-digital simulation to solve a kinetics problem with the KDA package. Therefore, two questions of interest are:

1. Is there an advantage to using one type of computer?
2. How does one determine which computer to use for specific problems?

The answer to the first question is there is an economic "break-even" point (see Figure 16) that governs the selection of a hybrid computer over a digital computer or vice versa. This "break-even" point is created when the simulation of the kinetic model requires the solution of a set of differential equations and the digital cost per optimization run is in excess of the equivalent hybrid cost.

A hybrid solution is practical when the hybrid economic advantage during the production phase of a kinetic study offsets and surpasses the deficit encountered during the problem preparation phase. Recalling previous discusions to perform a kinetic study using the Kinetic Data Analysis package the analog programming task is superimposed on the normal preparations required for an all-digital study. This creates an obvious hybrid deficit which combines with hybrid cost advantage during the execution of the optimization program to create an economic "break-even" point.

The economics associated with the hybrid versus all-digital question should be considered carefully because sufficient savings can be realized by making the correct decision. For example, a recent hybrid versus all-digital economic study for a reactor control problem[1] indicated that a large scale hybrid computer had approximately a 20:1 time and 40:1 cost advantage over large scale, third generation digital computers (e.g. $1,200 per hour computation center rate), and a 60:1 hybrid time advantage for the solution of the "Monsanto Benchmark Problem" has been reported in the literature.[6]

The hybrid cost advantage is directly related to the average computation time required to simulate a data set or experiment. The analog computer, typically requires 10–20 milliseconds to simulate one data set, which is independent of problem complexity. The time required for the equivalent digital simulation is a function of the speed of the digital computer, the number of equations, their degree of nonlinearity, and the integration algorithm. The influence of the digital integration algorithm on this situation is minor since the analog computer can be "speeded-up" more readily than the algorithm.

The answer to the question of how one determines the answer to the all-digital or hybrid question is very difficult due to lack of information. However, based on information obtained from several hybrid optimization studies performed on EAI 8900 Hybrid Computers, it was possible to derive some "rules-of-thumb" or guidelines. These relationships, which are based on a variety of studies involving up to twenty-six optimization variables, are admittedly crude.

The time required to execute one hybrid optimization run, including detailed printouts and tape manipulation, can be estimated using:

$$T_H \simeq 3 \cdot NOV \cdot NDS/100 \tag{15}$$

where

$T_H$ = time per hybrid optimization run, minutes
NOV = total number of optimization variables
NDS = total number of data sets

An approximate relationship to determine the equivalent time, $T_D$, for a digital optimization is:

$$T_D \simeq NOV \cdot NDS \cdot DST/1500 \tag{16}$$

where DST is the average number of milliseconds required to simulate one data set. This relationship does not include the time required for on-line I/O operations, which are not important if a competitive hybrid/digital situation exists.

A crude economic plot, see Figure 16, may be obtained from the equations:

$$C_H = C_H^P + (R_H \cdot T_H + C_H^E)\ NOR \tag{17}$$

and

$$C_D = C_D^P + (R_D \cdot T_D + C_D^E)\ NOR \tag{18}$$

where

$C_H$, $C_D$ = total hybrid and digital simulation costs
$C_H^P$, $C_D^P$ = estimated hybrid and digital preparation costs
$R_H$, $R_D$ = hybrid and digital computer rates
$C_H^E$, $C_D^E$ = engineering costs per optimization run
NOR = estimated number of optimization runs

The engineering costs associated with the execution and analysis of the optimization runs, $C_H^E$ and $C_D^E$, are not necessarily identical. For example, in the illustrative problem, four sets of four hybrid optimization runs (NOR = 16) were required and the engineering effort was four man days. An all-digital study could have required as long as, say, sixteen days to execute on a "slow" digital computer and required, say, eight man days of engineering.

The application of the above mentioned economic analysis to the "Monsanto Benchmark Problem" indicated that the "break-even" point was slightly less than thirteen optimization runs. Since the problem solution required sixteen optimization runs, the economics were only slightly in favor of a hybrid solution. However, a significant hybrid advantage was indicated if additional work was required. For example investigation of alternative mathematical models or analysis of additional experimental data.

## CONCLUSIONS AND COMMENTS

The present version of the Kinetic Data Analysis package has, based on limited customer utilization in

EAI Computation Centers, proven to be both an efficient and an economic means of performing both hybrid and all-digital studies. For example, the time required to obtain the all-digital optimization program has been one man-day or less for small- to medium-sized Kinetic Data Analysis studies.

Of greater significance, however, is the fact that this work has proven the practicality of hybrid applications software. It can be used as an effective tool to solve frequently oocurring problems on a routine basis with significant reductions in cost and problem preparation time. Therefore, the development of general purpose packages to solve specific classes of problems on hybrid computers would seem to be a fruitful area for future work.

## APPENDIX A

*Derivation of alternative reaction rate constant equation*

Defining the Arrhenius equation as:

$$K = A \cdot EXP\ (-B/T) \tag{1}$$

and a mid-range absolute temperature as

$$1/T_R = (1/T_H + 1/T_L)/2 \tag{2}$$

where

| | | |
|---|---|---|
| A, B | = | Arrhenius coefficients |
| $T_H$ | = | Maximum absolute temperature |
| $T_L$ | = | Minimum absolute temperature |
| $T_R$ | = | Reference absolute temperature |
| T | = | Absolute temperature, $T_L \leq T \leq T_H$ |
| K | = | Reaction rate constant |

one obtains:

$$K_H = A \cdot EXP\ (-B/T_H) \tag{3}$$

$$K_L = A \cdot EXP\ (-B/T_L) \tag{4}$$

$$K_R = A \cdot EXP\ (-B/T_R) \tag{5}$$

Combining equations 3 and 4 and equations 1 and 5 yields:

$$LN(K_{HL}) = LN\left(\frac{K_H}{K_L}\right) = B\left(\frac{1}{T_L} - \frac{1}{T_H}\right) \tag{6}$$

and

$$LN\left(\frac{K}{K_R}\right) = B\left(\frac{1}{T_R} - \frac{1}{T}\right) \tag{7}$$

which can then be combined to obtain

$$K = K_R(K_{HL})^{\beta} \tag{8}$$

and

$$\beta = (1/T_R - 1/T)/(1/T_L - 1/T_H) \tag{9}$$

Note that the range of $\beta$, based on equation 2, is $\pm \frac{1}{2}$ and the original Arrhenius coefficients in terms of $K_R$ and $K_{HL}$ are:

$$B = LN(K_{HL})/1/T_L - 1/T_H) \tag{10}$$

and

$$A = K_R \cdot EXP\ (+ B/T_R) \tag{11}$$

## REFERENCES

1 A CARLSON
*Hybrid simulation of an exchanger/reactor control system*
Presented at the Tech Conf on Process Control May 1968 Edmonton Alberta Canada

2 C GIESE
*Determination of best kinetic coefficients of a dynamic chemical process by on-line digital simulation*
Simulation Vol 20 1967 141

3 H H HARA R A NESBIT P E PARISOT
*A hybrid program for the solution of the Monsanto Benchmark problem*
Presented at Nat A I Ch E Meeting Columbus Ohio May 1966

4 A HARKINS
*The use of parallel tangents in optimization*
Chem Engr Prog Sym Series 60 35 1964

5 L LAPIDUS Y BARD
*Kinetic analysis by digital parameter estimation*
Catalysis Review Vol 2 67 1968

6 R A NESBIT H H HARA P E PARISOT
*Experiences with hybrid computer solution for kinetics parameters search problem*
Presented at Central States Simulation Council Meeting St Louis Missouri Jan 1966

7 P E PARISOT
*Parameter search for kinetic models utilizing the hybrid computers*
Presented at Midwest Simulation Council Meeting Aug 1965 Pittsburgh Pa

8 P E PARISOT L E FRANK V N SCHRODT

*Computer solution of sets of non-linear differential equations*
Presented at ACS Meeting Houston Texas Dec 1963

9 J B SCARBOROUGH
*Numerical mathematical analysis*
Johns Hopkins Press Balto 1958 4th ed 432-438

10 T J WILLIAMS
*Computer simulation of chemical reactions*
Chem Engr News Vol 20 1962 88

# The extended space technique for hybird computer solution of partial differential equations *

*by* DONALD J. NEWMAN and JON C. STRAUSS

*Carnegie-Mellon University*
Pittsburgh, Pennsylvania

## INTRODUCTION

The rapid solution of partial differential equations (PDE) has been a subject of increasing interest in recent years. This interest in partly due to advances in areas of technology which require the solution of PDEs, but is primarily due to the need to apply modern optimization and identification techniques to the spatially continuous systems that are best modeled by PDEs. The parallel organization of the analog subsection of a hybrid computer facilitates extremely rapid solutions of complicated systems of ordinary differential equations (ODEs). Therefore, techniques to find a system of ODEs that can be solved to obtain a rapid approximate solution to a PDE on the hybrid computer have become the subject of intensive investigation.

As digital computers have become faster and their memories larger, interest in symbol manipulation techniques has also increased, and advances have been made in the capabilities of computers to perform manipulative tasks once considered impractically large. The Galerkin technique for transforming a PDE into a system of ODEs has been known for some time, but for more than a crude solution of simple, linear problems, the quantity of algebra is so large that until recently this method has not been considered as a practical technique. However, the technolgoy has progressed to the point where the large quantity of algebra no longer prevents accurate solutions of both linear and nonlinear problems.

The Galerkin method employs an assumed solution consisting of a sum of time weighted spatial funtions; this separable form is similar to that used in the analytical technique for solution of linear PDEs commonly known as the separation of variables method. Each spatial function in the separable form is called a mode, and these modes are assumed to be known functions selected to satisfy the boundary conditions. The Galerkin method yields one ODE for each mode; the solution of the resulting system of ODEs yields the time varying weighting coefficients of the modes.

Recent investigation of the errors in assumed many-mode solutions of PDEs has led to the discovery that, while for the first few modes the Galerkin method is very effective, its performance for many-mode solutions is not satisfactory. The Galerkin method with small numbers of modes has been demonstrated to give more accurate solutions than other methods for the same number of ODEs.[1] If even more accurate solutions are required, more modes can be introduced into the solution, but the Galerkin method fails to produce results with any significant increase in accuracy for these multi-mode solutions. Although the Galerkin method has been shown to be convergent,[2] advances in symbol manipulation capability have shown that the method is limited in accuracy in practice by the extremely slow rate of convergence. Therefore, a new

* This work was supported by National Science Foundation Grant No. GJ-179. This paper was abstracted from the dissertation of D. J. Newman,[9] submitted in partial fulfillment of the requirements for a Ph.D. in Electrical Engineering from Carnegie-Mellon University.

technique that is effective for multi-mode solution is needed.

In this paper, a technique designed to meet this need, the extended space technique, is described and demonstrated. After a description of the PDE and the notion of assumed modes, a review of the Galerkin method introduces a thorough tutorial on the nature of the approximation errors. The linear problem with polynomial modes is used to further explain the slow convergence of the Galerkin method and to explain how the extended space technique overcomes this defect. Formal notation is introduced to make the technique applicable to the nonlinear problem. Finally an example problem is presented with comparative results based on an analytic solution.

A review and comparison of other hybrid methods is presented in a previous paper by the authors.[3] A more thorough explanation of the Galerkin method and its relationship to other assumed mode methods is available from a review article by Finlayson and Scriven,[4] the Ph.D. dissertation of D. J. Newman,[9] and a recent tutorial article by R. Vichnevetsky.[10]

### *Nonlinear partial differential equation*

The form of the PDE of interest is given in (1) where u(x,t) is the dependent function of independent variables x and t, P is a nonlinear partial differential operator with respect to x, and f is a forcing function.

$$\frac{\partial}{\partial t} u(x, t) = P[u(x, t)] + f(x, t) \tag{1}$$

The solution to this problem must satisfy an initial condition in t and homogeneous boundary conditions in x on the interval [0, 1]. (The [0, 1] interval is chosen for notational convenience only; the solution so obtained may be scaled to any other interval. Brackets are used to denote "operates on," and parentheses are used to denote that the value "depends on.") Thus (1) is an initial value problem in t, and retention of this initial value character in the system of ODEs to be obtained is desirable. The PDE form given in (1) appears to include only a limited number of PDEs, but through proper problem formulation a wide class of problems can be solved by simultaneous solution of PDEs of this form.

### *Assumed modes*

An approximate solution v(x,t) to (1) is proposed in the separable form of (2).

$$v(x, t) = \sum_{i=1}^{n} c_i(t)\, h_i(x) \tag{2}$$

The assumed spatial modes $h_i(x)$ are preselected to satisfy the orthogonality conditions of (3) and the spatial homogeneous boundary conditions on the solution to (1).

$$\int_0^1 h_i(x)\, h_j(x)\, dx = \begin{cases} 0 & i \neq j \\ ||h_i|| & i = j \end{cases} \tag{3}$$

Since the boundary conditions are homogeneous, v(x,t) also satisfies the spatial boundary conditions. A previous paper by the authors[3] removes the restriction to homogeneous boundary conditions, but it is retained in this paper to simplify the presentation. The The $c_i(t)$ functions are weighting functions for the assumed modes.

Subject to the conditions stated above, the selection of the modes depends on the problem knowledge of the solution, and computational convenience. If specific regions of the space differ in such a way that the solution has different characteristics there or very high accuracy is required, the problem should be subdivided into regions. The algebra for each region is a separate problem, but the resulting ODE systems are interdependent. A description of the regionalization problem is presented in Reference 9.

The $c_i(t)$ functions must be determined to give as nearly as possible the best solution to the PDE in (1) for the given modes of (2). The best approximation to the solution is one which matches the modal expansion of the exact solution u(x,t) for each mode in the approximate solution v(x,t). If u(x,t) is replaced in (1) by v(x,t), a residual function R(x,t) must be introduced to preserve the equality as shown in (4).

$$\sum_{i=1}^{n} h_i \frac{d}{dt} c_i = P\left[\sum_{i=1}^{n} c_i h_i\right] + f + R \tag{4}$$

The approximate solution v(x,t) is an exact solution to equation (4), but the intent is to solve equation (1) which differs from (4) by an additional forcing function R. Analyzing the difference between u(x,t) and v(x,t) is equivalent to analyzing the effect of adding the residual function R to the PDE.

### *Galerkin's approach*

The residual R in (4) is determined by the choice of the weighting functions $c_i(t)$ in the approximate solution v(x,t). Galerkin suggested in 1915[5] an approximation method based on orthogonalizing the residual with respect to the assumed modes; this orthogonality requirement is described by the n equations in (5).

$$\int_0^1 R(x, t)\, h_i(x)\, dx = 0 \qquad i = 1, 2 \ldots n \quad (5)$$

Galerkin does not give any justification for this method except to say that it is related to the work of Ritz.[6] However, in addition to the strong intuitive appeal, it is easily shown that the orthogonality condition of (5) can be obtained by minimizing the integral of the residual squared with respect to the time derivatives of the $c_i(t)$. This strong relationship to the variational methods of Ritz has led some investigators to refer to this method as the Ritz, Galerkin method.

Substituting (4) into (5) and employing the orthogonality conditions in (3) yields the ODEs in $c_i(t)$ given by (6).

$$||h_i|| \frac{d}{dt} c_i = \int_0^1 \left\{ P\left[ \sum_{i=1}^{n} c_i h_i \right] + f \right\} h_i dx \quad (6)$$

In this paper, the $c_i(0)$ are chosen to give a least squares fit of v(x,0) in (2) to the initial condition on u(x,t) in (1). Thus the $c_i(t)$ functions are determined from ODE initial value problems.

This approach can be generalized to any number of spatial variables as shown by Stacey.[7]

*Where does the residual go?*

To be sure, the residual does not vanish for most PDEs and most finite sets of modes. The expressions in (5) ensure that the residual is orthogonal to the $h_i(x)$ functions, hence the residual is not composed of the modes that are in the approximate solution to the problem. However, since the $h_i(x)$ must satisfy the boundary conditions, they do not form an appropriate basis for R and hence determination of $c_i(t)$ as in (6) does not minimize the residual in the most appropriate subspace.

A more useful form for investigating the residual is easily obtained by solving (4) for R and combining with (6) to obtain (7).

$$R(x, t) = \sum_{i=1}^{n} \frac{h_i}{||h_i||} \int_0^1 \left\{ P\left[ \sum_{i=1}^{n} c_i h_i \right] + f \right\} h_i dx - P\left[ \sum_{i=1}^{n} c_i h_i \right] - f \quad (7)$$

An analysis of (7) reveals that the residual must come from those parts of P[v(x,t)] and f(x,t) which are orthogonal to $h_i$ (x). The conclusion is that R acts as a forcing function composed of components of P[v(x,t)] and f(x,t) that are orthogonal to $h_i(x)$.

Is this effect good or bad?

With respect to f(x,t) even if the effect is not good at least the effect can be evaluated in terms of the physical problem. In short, f(x,t) might as well be assumed to be a function described by (2), and if certain properties of f must be considered in problem, modes characterizing these properties may be carried in the solution. This is quite tenable if f(x,t) obeys the boundary conditions, and equally impossible if f(x,t) does not.

With respect to P[v(x,t)], the effect is not immediately clear in terms of the physical problem. For modes which are not themselves solutions of the unforced problem (not natural modes), the effective forcing function contributed by R with components that are not in the solution can have effects on the solution. In the Galerkin method these effects emerge as errors in the approximate solution v(x,t) in addition to the error due to the omission of modes that are in u(x,t). These errors are caused by errors in the $c_i(t)$ functions and do not disappear very rapidly when more modes are added such as may be done for f(x, t).

Evidently the effects of the residual on the solution can be quite pronounced when mode and nonmode functions interact as may happen if nonnatural modes are employed.

*A special case*

The two-point boundary value problem with a second order linear PDE is a meaningful case to study. Since the object of this section is to examine the nature of the residual generated by the Galerkin method, the discussion is made more clear by assuming f(x,t) = 0 and by employing simple polynomial modes.

The modes for this two-point problem are required to satisfy the condition that $h_i(x)$ equal zero at the ends of the solution interval [0,1] for i = 1, 2, 3 ...,n. Actually more general conditions involving derivatives of $h_i(x)$ can be used as shown later in an example problem, and a still wider class of boundary conditions can be used as described in Reference 3. However, these conditions are simple and serve to demonstrate the principles involved.

The simplest polynomial modes that satisfy these boundary conditions are given in (8).

$$\bar{h}_i(x) = x^i(x - 1) \qquad i = 1, 2 \cdots, n \qquad (8)$$

The bar on the $\bar{h}_i(x)$ indicates that these functions are not orthogonal, but they are independent. The n orthogonal functions $h_i(x)$ defined in (3) are readily generated from the $\bar{h}_i(x)$ by the Gram-Schmidt procedure.

In order to determine the composition of the residual, $P[h_i(x)]$ must be examined to determine which components are orthogonal to all of the $h_i(x)$. For this purpose, $\bar{h}_i(x)$ is an adequate substitute for $h_i(x)$ and considerably simplifies the discussion. Since P is a linear combination of derivative operators, $P[\bar{h}_i(x)]$ could not contain any powers of x greater than i + 1 but could have any lower term including a constant term. In fact an adequate basis for $P[\bar{h}_i(x)]$ includes in addition to the $\bar{h}_i(x)$ functions two functions 1 and x that do not satisfy the boundary conditions. Therefore, the residual must be composed of a linear combination of 1 and x, and the Galerkin solution for this special case has an effective forcing function of the form ax + b.

Introducing such an extraneous function or alternatively ignoring such a function if it were part of f(x,t) does not seem to be reasonable. Ostensibly the residual in the PDE is due to the omission of modes of higher degree from the approximate solution; however, such a residual would not be a function ax + b but would contain all modes especially those of the highest degree included in the solution.

*The extended space technique for the special case*

This technique extends the space of functions being considered for the solution of the second order linear PDE to include functions, $h_{n+1}(x)$ and $h_{n+2}(x)$, which are used to absorb the residual and reduce the error in the coefficients $c_i(t)$; however, these functions are not included in the actual approximate solution v(x,t). In the extended space technique, the residual is not part of P[v(x,t)]; instead, the residual consists of functions that are not part of the approximate solution and cannot be generated in the PDE from the approximate solution. The addition of sufficient amounts of $h_{n+1}(x)$ and $h_{n+2}(x)$ to remove the ax + b component from the residual reduces the error in the coefficients for the modes.

The expression given in (9) is substituted into the PDE instead of the approximate solution v(x,t) to generate the extended space residual $R_e(x,t)$.

$$\sum_{i=1}^{n+2} h_i(x)\, c_i(t) = v(x, t) + h_{n+1}(x)\, c_{n+1}(t) + h_{n+2}(x)\, c_{n+2}(t) \qquad (9)$$

Two functions which are orthogonal to the $h_i(x)$, i = 1, 2,. . ., n + 2 can be found from 1 and x and are denoted $g_1^0(x)$ and $g_2^0(x)$. These functions with $h_i(x)$, i = 1, 2. . .n form a basis for P[v(x, t)]. The $g_i^0(x)$ functions are employed in (10a) to give two linear algebraic equations which when solved simultaneously with the n linear ODEs in (10b) determine the coefficients $c_i(t)$ in v(x, t).

$$\int_0^1 g_j^0(x)\, R_e(x, t)\, dx = 0 \qquad j = 1, 2 \qquad (10a)$$

$$\int_0^1 h_i(x)\, R_e(x, t)\, dx = 0 \qquad i = 1, 2, \cdots, n \qquad (10b)$$

The two equations in (10a) insure that the residual will not have 1 and x as a basis. The equations in (10b) are essentially the same as those in (5) and insure that the residual is orthogonal to the modes. The conditions in (10) are necessary for the minimization of the integral R squared in the subspace with basis $g_1^0$, $g_2^0$, $h_i$ (i=1,...,n). It has been demonstrated that this is a more appropriate subspace for the description of R than that with $h_i$ (i=1,...,n) alone as a basis. It should therefore be expected that the extended space technique give better results than the Galerkin method.

A close examination of P[u(x,t)] (u(x,t) is the exact solution) compared to P[v(x,t)] indicates why the extended space technique does give better results. P[u(x,t)] can be broken into three important parts: P[v(x,t)] is one part, a part which has the same basis as P[v(x,t)] but is generated by functions in u(x,t) that are not in v(x,t) is a second part, and a part which has a basis different from P[v(x,t)] is a third. Because the third part has no effect on (10), it cannot cause any error in the coefficients $c_i(t)$, but the second part can. Because the second part is generated by functions not in v(x,t), it does not appear in (4). Ideally the residual should be this second part, but since the second part is functionally indistinguishable from the first part, the ideal residual cannot be produced. The extended space technique alleviates the errors caused by the absence of the second part for two reasons: (1) the extension functions $h_{n+1}(x)$ and $h_{n+2}(x)$ do generate some of the second part; (2) since the residual is composed of these extension functions, the effective forcing function is not composed of only the $g_1^0(x)$ and $g_2^0(x)$ which should have been cancelled out of (4) by the second part. Particularly in this special case, linear P with polynominal modes where the greatest interaction is between adjacent modes, the majority of the effect of the second part of P[u(x,t)] is absorbed by these two mechanisms.

*A generalization of the technique*

The extended space technique can be generalized to cover a nonlinear PDE with m boundary conditions where the solution employs nonpolynomial modes. Unfortunately, the effect of the technique on the residual and the error in the coefficients cannot be readily examined under these general conditions.

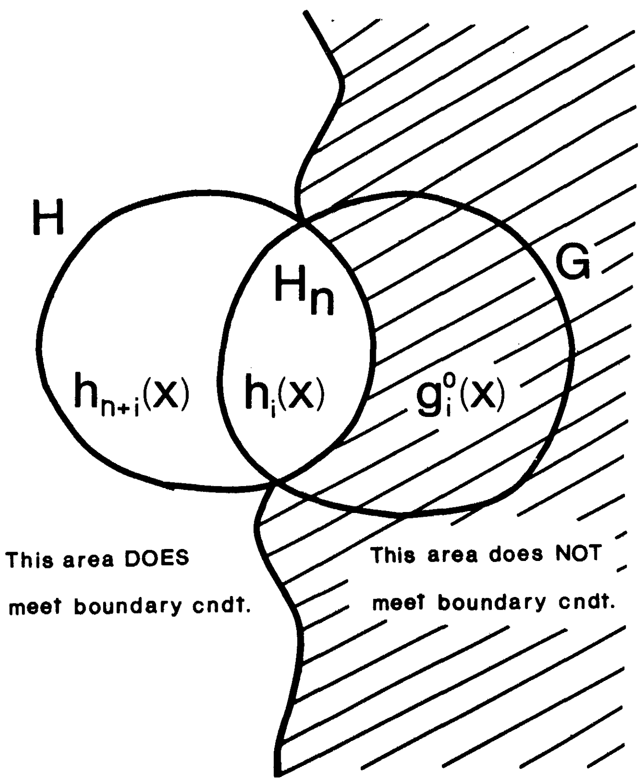

H
G
$H_n$
$h_{n+i}(x)$
$h_i(x)$
$g_i^0(x)$
This area DOES meet boundary cndt.
This area does NOT meet boundary cndt.

Figure 1—Function spaces

In order to proceed with the description, a more general notation is required: G is the set of functions that are desired as modes and functions to fit the forcing function, f(x,t). G has an orthogonal basia of n + m functions denoted $g_i(x)$. $H_n$ is a subset of G such that all functions in $H_n$ satisfy the m boundary conditions. H is an extension of $H_n$ outside of G, but all of the functions in H also satisfy the boundary conditions. H also has an orthogonal basis of n + m functions denoted $h_i(x)$, and the first n of these funtions are in $H_n$. In addition m functions denoted $g_i^0(x)$ are defined to be orthogonal to all functions in H and along with the $h_i(x)$ in $H_n$ form a basis for G. The relationship of these sets of functions is shown pictorially in Figure 1.

The approximate solution retains the form given in (2), but the residual $R_e$ is given by (11).

$$\sum_{i=1}^{n+m} h_i \frac{d}{dt} c_i = P\left[\sum_{i=1}^{n+m} h_i c_i\right] + f + R_e \quad (11)$$

The system of equations that are solved to determine $c_i(t)$ are given by (12) and are derived from orthogonality conditions as in (10).

$$0 = \int_0^1 \left\{ P\left[\sum_{i=1}^{n+m} h_i c_i\right] + f \right\} g_j^0 \, dx \qquad j = 1, 2, \cdots, m \quad (12a)$$

$$||h_j|| \frac{d}{dt} c_j = \int_0^1 \left\{ P\left[\sum_{i=1}^{n+m} c_i h_i\right] + f \right\} h_j \, dx \qquad j = 1, 2 \cdots n \quad (12b)$$

Equations (12a) are a nonlinear algebraic system and (12b) are a nonlinear ODE system.

*A linear PDE problem*

The study of heat transfer within a solid is an interesting problem in connection with this work because the surface conditions give rise to a two-point boundary value problem. Problems of this nature are encountered in heat exchangers where metallic fins are cooled by a forced flow of a fluid. In this example problem, the linear diffusion equation shown in (13) is used to represent one dimensional heat flow within the metal fin.

$$k \frac{\partial^2 u(x, t)}{dx^2} = \mu \frac{\partial u(x, t)}{\partial t} \quad (13)$$

In (13) u(x,t) is the temperature, k the conductivity and $\mu$ the heat capacity of the metal.

Newton's law of cooling shown in (14) is used for the boundary condition at the fin surface cooled with the fluid at temperature s(t).

$$\frac{\partial u(x, t)}{\partial x} = \alpha(u(x, t) - s(t)) \tag{14}$$

This problem is a linear PDE problem with a linear differential boundary condition.

The problem is chosen as an example because it has an analytical solution. The solution is evaluated and used for comparisons of the accuracy of two, three and five mode assumed mode solutions employing the Galerkin technique and with two and three mode solutions employing the extended space technique.

*A metal fin*

The example problem deals with a fin of metal uniform in thickness which is cooled by water on both sides as shown in Figure 2. The initial temperature (100°) is uniform throughout the cross section, and cooler water (0°) begins to circulate by the fin at time zero. The problem has symmetry so that only half the fin must be considered in the problem.

The water-metal surface is assumed to obey Newton's law of cooling which requires that the rate of transfer of energy through a boundary be proportional to the temperature difference across that boundary. The rate of transfer is proportional to the derivative of temperature within the metal at the surface. The difference in temperature across the boundary is the difference between the temperature within the metal at the surface and the water temperature, a function of time s(t). In (14), the proportionality constant $\alpha$ is assumed to be equal to one for simplicity, and s(t) is assumed to be the step function given in (15) which is chosen so that the problem will have a simple analytic solution.

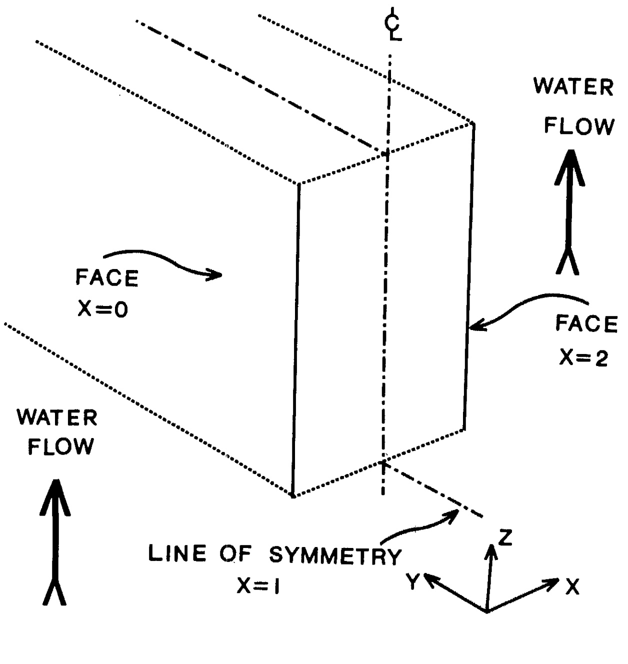


Figure 2—Metallic fin

$$s(t) = \begin{cases} 100 & t = 0 \\ 0 & t > 0 \end{cases} \tag{15}$$

While the surface provides one boundary condition, symmetry provides another since the derivative of temperature must be zero on the axis of symmetry. The complete PDE problem is given in (16) where k = 1/10 and $\mu$ = 1.

$$\frac{1}{10}\frac{\partial^2 u(x, t)}{\partial x^2} = \frac{\partial u(x, t)}{\partial t}$$

$$\left.\frac{\partial u(x, t)}{\partial x}\right|_{x=1} = 0 \tag{16}$$

$$\left.\left(\frac{\partial u(x, t)}{\partial x} - u(x, t)\right)\right|_{x=0} = 0$$

$$u(x, 0) = 100$$

The analytical solution to this problem is found by classical separation of variables, is quite complicated, and is not harmonic in nature. The frequencies of the sine components vary according to the solutions of $\omega = \alpha \cot(\omega)$. In a sample problem given by Lebedev, Skal'Skaya and Uflyand,[8] an answer is given which is presumably an exact answer for this problem. Actually their expression is a close approximation to the exact solution with an accuracy of better than .01 percent for $\alpha = 1$.

*Modes and the ODE system*

The modes for this problem are chosen by application of the method for homogeneous differential boundary conditions presented in Reference 3. A simple polynomial family $g_i(x) = ix - x^i$ is employed as the set G which satisfies $g_i'(1) = 0$. The modes are integrated with the boundary condition at $x = 0$ applied to determine the integation constant, and the $\bar{h}_i(x)$ shown in (17) are produced.

$$\bar{h}_i(x) = (i + 1)(x + 1) - x^{i+1} \quad i = 1,2\ldots n \qquad (17)$$

The modes $h_i(x)$ are obtained by orthogonalization of $\bar{h}_i(x)$ and are shown in Table I.

TABLE I—Orthogonal modes for linear PDE problem

$$h_1(x) = - X \uparrow 2 + 2X + 2$$

$$h_2(x) = - X \uparrow 3 + (691/432)X \uparrow 2 - (43/216)X - (43/216)$$

$$h_3(x) = - X \uparrow 4 + (7932/3905)X \uparrow 3 - (58089/54670)X \uparrow 2 + (857/27335)X + (857/27335)$$

$$h_4(x) = - X \uparrow 5 + (148725/59152)X \uparrow 4 - (30469/14788)X \uparrow 3 + (200593/354912)X \uparrow 2 - (1129/177456)X - (1129/177456)$$

$$h_5(x) = - X \uparrow 6 + (432358/143745)X \uparrow 5 - (31431/9583)X \uparrow 4 + (2431444/1581195)X \uparrow 3 - (1266961/4743585)X \uparrow 2 + (6746/4743585)X + (6746/4743585)$$

The ODE systems for two, three and five modes employing the Galerkin technique are obtained by the application of equation (6). The derivatives of $c_i(t)$ are linear functions of the $c_i(t)$, and the coefficient matrices of the equations are given in Table II, The ODE systems for two and three modes employing the extended space technique are obtained by substitution of (12a) into (12b) to eliminate the highest two $c_i(t)$ Again the derivative functions are linear, and the coefficient matrices are given in Table III.

TABLE III—Coefficient matrices for extended space technique

| | |
|---|---|
| -.0740726 | .000426177 |
| .142572 | -1.174004 |

| | | |
|---|---|---|
| -.0740739 | .000427785 | $-1.799_{10}-5$ |
| .1431100 | -1.174666 | .0741377 |
| -.301688 | .371627 | -4.15921 |

*Numerical results*

The simple boundary condition employed in this problem to facilitate obtaining an analytical solution presents some severe difficulties in obtaining a good fit to the initial condition. The modes must satisfy this unrealistic boundary condition which imposes a steep slope at $x = 0$ where the initial condition is flat. Figure 3 shows the solution fit to the initial condition for two, three and five modes for both techniques. Even at five modes the fit is not entirely satisfactory; however, for small numbers of modes, the analytical solution suffers from the same defect. This is the cost that must be paid to obtain an analytical solution for comparison.

Figure 4 shows the solution at times of five, 20 and 100 seconds for both techniques. The solutions, exact, two, three and five mode are indistinguishable on a graph of this scale. The hybrid solution also produces identical results and the analog block diagram for this problem with three modes is shown in Figure 5.

TABLE II—Coefficient matrices for Galerkin method

| | | | | |
|---|---|---|---|---|
| .074074 | $4.2867_{10}-4$ | $1.8292_{10}-5$ | $1.4610_{10}-6$ | $-1.5616_{10}-7$ |
| .14341 | −1.1769 | $7.7397_{10}-3$ | $-3.8192_{10}-3$ | $7.5717_{10}-5$ |
| − .30674 | .38796 | −19.084 | .020270 | − .034980 |
| .71994 | −5.6258 | .59565 | −9.4932 | .033540 |
| −1.8134 | 2.6283 | −24.223 | .79036 | −17.439 |

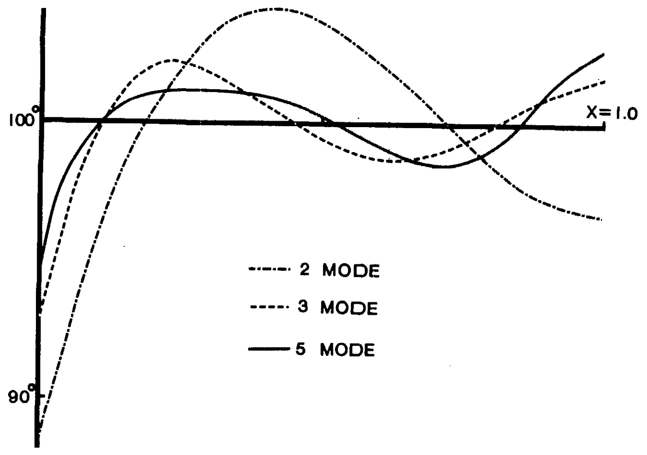


Figure 3—Linear PDE problem T = 0

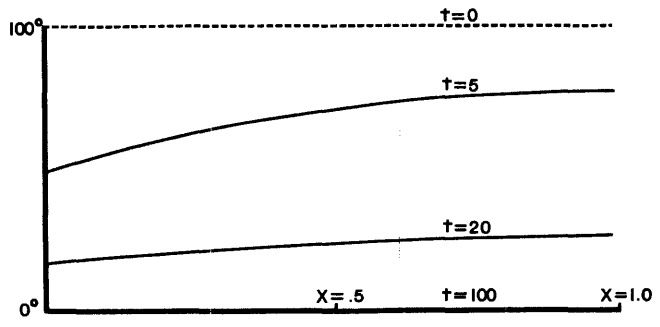


Figure 4—Linear PDE problem solution

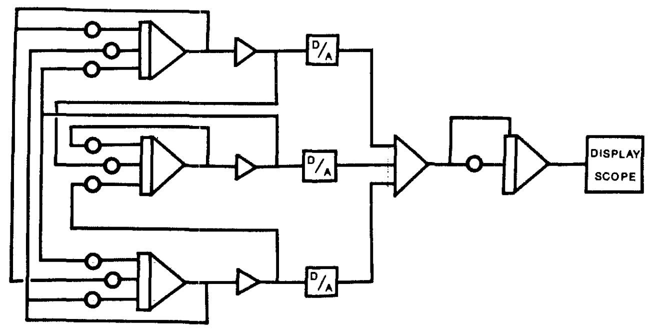


Figure 5—Linear PDE problem analog diagram

In order to compare the accuracies of the different solutions, errors for cross sections at five seconds are chosen because the five second cross section has the greatest error and because at five seconds the analytic solution is sufficiently convergent to give an accurate basis for comparison. Figure 6 shows the error curves on a greatly magnified scale (full scale is .15 to .2 per-

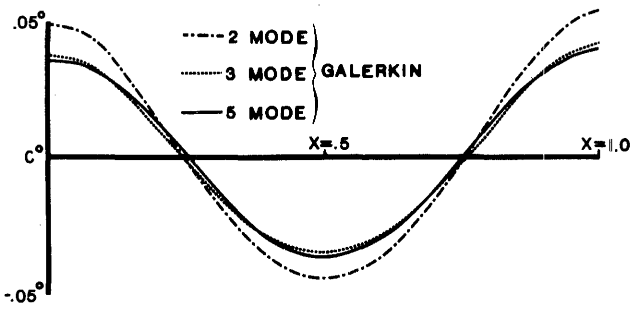


Figure 6—Galerkin method error T = 5.0

cent of the solution) for the three assumed mode solutions using the Galerkin method. The improvement between the two and three mode solution is substantial but the five mode solution is disappointingly similar to the three mode solution. The error does not decrease nor does it change shape. Since in a five mode solution only the seventh and higher degree polynomials are excluded from the solution, the logical conclusion would be that the error would have three maxima and three minima; but since it does not, possibly an error has crept into a lower mode which is not diminishing to zero very rapidly. The analysis performed previously indicates that this is in fact the case and that even though this error does slowly diminish as the number of modes increases, the Galerkin method on linear problems leaves all the error in the modes that are part of the solution.

The extended space technique shows a dramatic improvement in the accuracy of the results for three modes. Figure 7 shows the error curves for the two and three mode solution; the three mode solution matches the analytic solution so well that five mode solution is not needed. The error shown for the three mode solution is so small that it is comparable to the errors in numerical integration of the ODE system and is only meaningful in the sense that it is a great improvement over the Galerkin method.

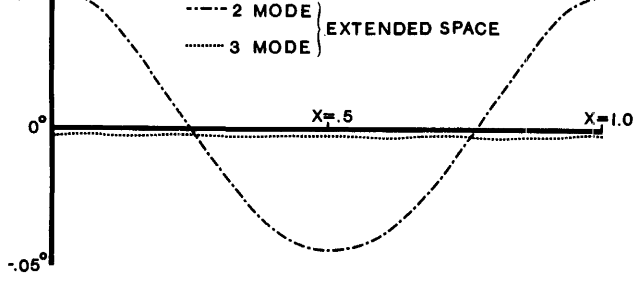


Figure 7—Extended space error T = 5.0

TABLE IV—Comparison of eigenvalues

| | Galerkin | | | Extended Space | |
|---|---|---|---|---|---|
| Analytic | 2-Mode | 3-Mode | 5-Mode | 2-Mode | 3-Mode |
| - .07402 | - .07402 | - .07403 | - .07403 | - .07402 | - .07402 |
| -1.1734 | -1.1770 | -1.1768 | -1.1742 | -1.1741 | -1.1738 |
| -4.1439 | | -19.084 | -9.4916 | | -4.1601 |
| -9.0810 | | | -17.028 | | |
| -16.000 | | | -19.499 | | |

TABLE V—Comparison of digital computation times

| | Galerkin | | Extended Space | |
|---|---|---|---|---|
| | Algebra | Integration* | Algebra** | Integration* |
| 2-Mode | 11 | 6 | 98 | 6 |
| 3-Mode | 34 | 7 | 234 | 7 |
| 5-Mode | 209 | 10 | | |

Times in seconds for IBM-360/65

* Does not include the time for the compilation of ODE derivative subprogram requiring about 30 seconds.

** Values corrected to remove estimated program compilation time which was not included in other timings.

The eigenvalues for the various solutions shown in Table IV indicate why the extended space technique produces such accurate results. For all solutions the first two eigenvalues match the eigenvalues obtained from the exact solution very well. The third eigenvalue for the Galerkin method is never very near the exact value even for five modes; however, the extended space technique produces an eigenvalue very near the exact solution with only three modes. In fact the extended space technique produces a much better eigenvalue for three modes than the Galerkin method does for five modes. Table V presents a comparison of the digital computation times to do the algebra necessary to prepare the ODE systems and times to do the numerical integration of the systems for the Galerkin method and the extended space technique. Computation time on the hybrid computer to solve the ODE system is the same for all cases and may be as small as 10 milliseconds on the Carnegie-Mellon University EAI-680/PDP-9 hybrid computer depending on the I/O device used to monitor the solution.

## CONCLUSIONS

The example problem has demonstrated how much of an improvement the extended space technique can be over the classical Galerkin method. Both the accuracy of the solution and the eigenvalues of the ODE system are better for the three mode extended space technique solution. However, this improvement is not obtained without some increased cost: the quantity of algebra that must be performed to determine the three mode extended space solution is about equal to the quantity to determine the five mode Galerkin solution. Even when this increased cost is considered, the extended space technique is superior because the three mode solution is better than the Galerkin five mode solution.

This technique is also applicable to nonlinear problems, but no experimental results are available at present. The nonlinear application has an additional complication: the simultaneous solution of nonlinear algebraic equations and a nonlinear ODE system is required. Work on a nonlinear problem is currently being done and results are expected to indicate comparable superiority over the Galerkin method for nonlinear problems.

## REFERENCES

1 W Z COLLINGS
*The method of undetermined functions as applied to nonlinear diffusion problems*
MME thesis Univ of Delaware 1962

2 M A KRASNOSEL'SKI
*Topological methods in the theory of nonlinear integral equations*
Pergamon-Macmillan 1964

3 D J NEWMAN J C STRAUSS
*Hybrid assumed mode solution of nonlinear partial differential equations*
Proc FJCC Vol 33 1968 575

4 B A FINLAYSON L E SCRIVEN
*The method of weighted residuals—A review*
Applied Mechanics Reviews Vol 19 No 9 Sept 1966 735

5 B G GALERKIN
*Rods and plates*
Vestn Inzhen i Tekh Petrograd 19 897-908 1915
Translation 63-18924 Clearinghouse Fed Sci-Tech Info

6 W RITZ
Über eine neue Methode zur Lösung gevisser Variations problem der mathemalischen physik J f reine u angewandte Mathematik, 1909

7 W M STACEY JR
*Modal approximations*
MIT Press Cambridge 1967

8 N N LEBEDEV IP SKAL'SKAYA Y S UFLYAND
*Problems in mathematical physics*
Translated by A R M Robson Pergamon Press Oxford 1966

9 D J NEWMAN

*Hybrid assumed mode solution of nonlinear partial differential equations*
Carnegie-Mellon Univ Pittsburgh Pa 1969 PhD thesis
Available from Univ Microfilms Ann Arbor Michigan

10 R VICHNEVETSKY
*Use of functional approximation methods in the computer solution of initial value partial differential equation problems*
IEEE Transactions on Computers Vol C-18 June 1969

# Extension and analysis of use of derivatives for compensation of hybrid solution of linear differential equations

*by* NELSON H. KEMP

*Wolf Research and Development Corporation*
West Concord, Massachusetts

## INTRODUCTION

When compared to continuous (analog) computation, hybrid computation is subject to two sources of error not associated with hardware, but caused by its logical nature. They are often referred to as the time (or transport) delay, and the reconstruction errors.

This time delay error is caused by the time taken for the digital computer to process the data sampled from the analog computer, before sending the updated results back to the analog. The reconstruction error results from the hold action of the digital-to-analog link: the updated value from the digital is sent to the analog and held fixed until the next updating, instead of being updated continuously.

The effect of these errors on the hybrid solution (as compared with a pure analog solution) is twofold. First, inaccuracies are introduced. Second, the hybrid solution may become instable and grow without bound, even though the correct solution is bounded or even decreases to zero.

To prevent instability and minimize error, hybrid computations utilize compensation techniques. The variables processed in the digital computer for use in the analog computer are calculated at some future time, by an extrapolation scheme, before being sent to the analog. Depending on the scheme used, this technique can have a beneficial effect on the accuracy and stability of the solution, for a given sampling interval.

There are a number of extrapolation techniques commonly used to achieve compensation. One such technique is that of multistep extrapolation, or digital filters, in which values of the variables at earlier time are used for extrapolation. A good discussion of this method is given by Mitchell.[1] He demonstrates its shortcomings for heavily damped systems, caused by the instability of the extraneous solutions introduced by use of values at earlier times. For each step back in time, one extraneous solution is introduced, and these solutions are instable for large enough sampling intervals. The popular three-step, or parabolic, extrapolation introduces two such solutions, and their amplitude increases with increasing damping, so that heavily damped systems require small sampling intervals for stability.

Some years ago, Miura and Iwata[2] suggested another technique of extrapolation. For solving differential equations, they used the derivative of each variable to extrapolate, rather in the manner of a Taylor series. The implementation suggested was to add to the output of an integrator a multiple of the input, the sum being the extrapolated value of the variable. Further use of this scheme, for undamped systems, was made by Gilbert[3] and Karplus[4,5] with several implementations suggested. Gilbert[3] analyzed the undamped system, using z-transforms. This extrapolation technique has the advantage of requiring either no backward steps, or only one, depending on the implementation, thus eliminating or reducing the number of extraneous solutions introduced. The result is a solution which is not only more accurate than the uncompensated hybrid solution, but can be more stable. This is in contrast to the use of multi-step methods, which improve the accuracy but reduce the stability compared to the uncompensated hybrid solution.

There is apparently only one published reference to the use of the method of Miura and Iwata for a damped second order system. Bekey and Karplus,[5] on pages 382-383 of Chapter 12, give some results of unpublished* work of Howe and Fogarty.[6] In this work, they extrapolate x and $\dot{x}$ by using 1.5 T times $\dot{x}$ and x respectively, where T is the sampling interval. They use an implementation where the extrapolation is performed in the analog computer, the extrapolated values are sampled by the digital computer, combined to give $\ddot{x}$, and then converted D to A and sent to the analog computer for integration. We can call this calculation of extrapolated values in the analog computer *analog* compensation. The analysis by z-transforms is based on a timing sequence in which the A to D sampling occurs *before* the D to A conversion of $\ddot{x}$. The result of this compensation scheme is two desirable solutions which have exponents whose error are of order $(\omega T)^2$, in contrast to error of order $\omega T$ for the uncompensated solution, where $\omega$ is the natural frequency. However, there are two extraneous solutions of the order $(\zeta\omega T)^{\frac{1}{2}}$, where $\zeta$ is the damping coefficient, in contrast to the single extraneous solution of order $\zeta\omega T$ for no compensation. Therefore, we see that in this case derivative compensation improves the accuracy, but it reduces the stability, compared to no compensation.

This situation can be improved if we change to what might be called *digital* compensation. Here, we sample x and $\dot{x}$, and do the extrapolations in the digital computer. This is the scheme used in the present report. For a damped system, it uses no backward time steps, instead of the one backward step inherent in the Howe-Fogarty implementation. Therefore, it has only one extraneous solution, of order $\zeta\omega T$, and is somewhat more stable than the uncompensated case because of a better numerical factor. The accuracy of the two desirable solutions is of the same order as those of Howe and Fogarty.

The same scheme as that given for digital compensation in this report can be obtained by the analog compensation method of Howe and Fogarty if they change the order of A to D sampling and D to A conversion and perform D to A *before* A to D. This may not be a desirable implementation because the transients set up by D to A may interfere with the values sampled A to D immediately thereafter.

* Prof. Fogarty kindly sent me a copy of this report, and the remarks in this paragraph are based on my analysis of Section 5 of the report.

The purpose of this report is to extend the use of derivatives for extrapolation, to apply the method to a damped second order system typical of control problems, to analyze the system by use of z-transforms, and to compare the analysis with hybrid calculations using both derivative compensation and multi-step compensation.

The extension of the derivative method, which is also referred to as Taylor series compensation, is in several directions. First, we not only correct x by using $\dot{x}$, but also by using $\ddot{x}$, since that derivative is also available. Second, we do not assume an extrapolation ahead by 1.5T, but carry along arbitrary constants which are then chosen to give greatest accuracy. The first order corrections are indeed found by this method to be 1.5T, providing a simple analytical derivation of this fact. The second order coefficient of $\ddot{x}$ may be chosen in several ways toenhance accuracy or stability.

The analysis is applied to a linear damped oscillator, forced by a control function which is a linear combination of x and $\dot{x}$. The oscillator is implemented on the analog computer, the control function on the digital computer.

The z-transform analysis yields formulas which can be used to predict the stability of both the compensated and uncompensated cases for any values of the parameters and sampling interval. Similar results are given for the three-step compensation scheme, and show it to be less stable.

A numerical test was made by implementing both schemes on a Beckman 2200/SDS 9300 hybrid computer. The hybrid calculations were compared with continuous calculations of the same system made on the analog computer. The superior accuracy and stability of the Taylor series method over the three-step method is clearly apparent in the strip chart results, as well as in the digital printouts.

## *Analysis*

### Continuous solution

The forced oscillator analyzed is defined by

$$x \times 2\omega\zeta\dot{x} + \omega^2 x = \omega^2\delta \qquad (2.1)$$

$$\delta = Kx_c - K(\tau\dot{x} + x) \qquad (2.2)$$

where K and $\tau$ are constant control parameters. The command input $x_c$ is taken to be a constant here, for ease of analysis. Further, only the simple initial conditions $x(0) = 0$, $x(0) = 0$ are considered, although other values bring only algebraic complication.

The exact continuous solution of this problem is simply obtained by transposing the variables on the right side and defining total frequency and damping by

$$\omega^2_T = \omega^2(1 + K), \quad 2\omega_T \zeta_T = 2\omega\zeta + \omega^2 K_T \quad (2.3)$$

The solution with zero initial conditions is then

$$x = \frac{Kx_c}{1+K}\left[1 - \frac{1}{2}\left(1 - \frac{i\zeta_T}{\xi^1_T}\right)e^{\lambda_{T1}t} - \frac{1}{2}\left(1 + \frac{i\zeta_T}{\zeta^T_1}\right)e^{\lambda_{T2}t}\right] \quad (2.4)$$

$$\zeta^1{}_T = (1 - \zeta^2{}_T)^{1/2}, \quad \lambda_{T1,2} = \omega_T(-\zeta^1{}_T \pm i\zeta_T{}^1)$$

where the $\lambda_{T1,2}$ are the roots of the characteristic equation

$$\lambda^2{}_T + 2\omega_T\zeta_T\lambda_T + \omega^2{}_T = \lambda^2{}_T + (2\omega\zeta + \omega^2 K_T)\lambda_T + \omega^2(1 + K) = 0 \quad (2.5)$$

## Hybrid difference Equations

The hybrid implementation considers the $\delta$ term as a control function which is calculated digitally while the left side of (2.1) is calculated continuously in the analog computer. Thus, between the sampling times nT and (n + 1)T, $\delta$ is held fixed at the value $\delta_{Pn}$ supplied to the analog at t = nT.

Therefore during this interval the analog solves

$$x + 2\omega\zeta\dot{x} + \omega^2 x = \omega^2\,\delta_{Pn} \quad (2.6a)$$

with initial conditions

$$x = x_n \qquad \dot{x} = \dot{x}_n \quad (2.6b)$$

The solution of (2.6) is

$$x = \left[\frac{\lambda_2(x_n - \delta_{Pn}) - \dot{x}_n}{\lambda_2 - \lambda_1}\right]e^{\lambda_1(t-nT)} + \left[\frac{\dot{x}_n - \lambda_1(x_n - \delta_{Pn})}{\lambda_2 - \lambda_1}\right]e^{\lambda_2(t-nT)} + \delta_{Pn}$$

$$\dot{x} = \lambda_1[\quad '' \quad]\,e^{\lambda_1(t-nT)} + \lambda_2[\quad '' \quad]\,e^{\lambda_2-(tnT)} \quad (2.7a, b)$$

where $\lambda_{1,2}$ are the roots of the free-vibration characteristic equation

$$\lambda^2 + 2\omega\zeta\lambda + \omega^2 = 0, \quad \lambda_{1,2} = \omega(-\zeta \pm i\zeta^1), \quad \zeta^1 = (1 - \zeta^2)^{1/2} \quad (2.8)$$

At t = (n + 1)T these are expressible in real form as

$$x_{n+1} = e^{-\omega\zeta T}\left[(x_n - \delta_{Pn})(\cos\omega\zeta^1 T + \zeta/\zeta^1 \sin\omega\zeta^1 T) + \dot{x}_n/\omega \sin\omega\zeta^1 T\right] + \delta_{Pn} \quad (2.9a)$$

$$\dot{x}_{n+1} = e^{-\omega\zeta T}\left[\dot{x}_n(\cos\omega\zeta^1 T - \zeta/\zeta^1 \sin\omega\zeta^1 T) - \omega(x_n - \delta_{Pn})\sin\omega\zeta^1 T\right] \quad (2.9b)$$

These two equations are difference relations between $x_n$, $\dot{x}_n$ and $x_{n+1}$, $\dot{x}_{n+1}$, with given $\delta_{Pn}$. Equations (2.7) show that the analog computer produces segments of forced damped vibrations between sampling times, each joined to the adjacent segments with continuous x and $\dot{x}$, but discontinuous x, because $\delta_{Pn}$ changes at each sampling time. The hybrid system solves the difference equations (2.9), as will we, but first $\delta_{Pn}$ must be specified in terms of x and $\dot{x}$ to model the digital part of the calculation.

## Taylor series compensation

The digital calculation of $\delta_{Pn}$, the value sent to the analog at time nT, can only depend on quantities sampled by the digital at previous sampling times. We will project x and $\dot{x}$ and take $\delta_{Pn}$ to be given by the projected values according to (2.2):

$$\delta_{Pn} = Kx_c - K(\tau\dot{x}_{Pn} + x_{Pn}) \quad (2.10)$$

The projections are accomplished from $x_{n-1}$, $\dot{x}_{n-1}$ by a Taylor series form

$$x_{Pn} = x_{n-1} + \ell T\dot{x}_{n-1} + kT^2 x_{n-1} \quad (2.11a)$$

$$\dot{x}_{Pn} = \dot{x}_{n-1} + hT\,x_{n-1} \quad (2.11b)$$

We have used as many terms as the available derivatives allow. The quantity $\dot{x}_{n-1}$ can be sampled and made available in the digital. The second derivative is calculated from the differential equation (2.6a)

$$x_{n-1} = -2\omega\zeta\,\dot{x}_{n-1} - \omega^2 x_{n-1} + \omega^2\,\delta_{P,n-1} \quad (2.12)$$

Equation (2.10)–(2.12) are the essence of the Taylor series compensation scheme proposed here. In contrast, a three-step scheme would project $\delta_{Pn}$ from previous $\delta$'s:

$$\delta_{Pn} = a_0\,\delta_{n-1} + a_1\delta_{n-2} + a_2\,\delta_{n-3} \qquad (2.13a)$$

where

$$\delta_{n-1} = Kx_c - K(\tau\,\dot{x}_{n-1} + x_{n-1}) \qquad (2.13b)$$

and similarly for $\delta_{n-2}$, $\delta_{n-3}$. This scheme goes back to $(n-3)T$, two steps further than (2.11).

In both cases the constants $\ell$, k, h, or $a_0$, $a_1$, $a_2$ are available to help improve the solution. For the three-step method, it is conventional to project to the time $(n + 1/2)T$, for which the values of the constants are

$$a_1 = -21/4,\ a_2 = 15/8,\ a_0 = 1 - a_1 - a_2 = 35/8 \qquad (2.14)$$

If we project (2.11) the same distance, we find

$$\ell = h = 3/2, \qquad k = 9/8 \qquad (2.15)$$

Instead we will carry the constants along, and choose their values on the basis of the resulting formulas.

The final form of $\delta_{Pn}$ comes by inserting (2.11) and (2.12) into (2.10) to obtain

$$\begin{aligned}\delta_{Pn} = Kx_c - K\{x_{n-1}\,&(1 - h\omega\tau\omega T - k\omega^2T^2)\\ &- \delta_{P,n-1}\,(h\omega\tau\omega T + k\omega^2T^2)\\ &+ \omega^{-1}\,\dot{x}_{n-1}[\omega\tau + (\ell - 2\zeta h\omega\tau)\omega T\\ &- 2\zeta k\omega^2T^2]\}\end{aligned} \qquad (2.16)$$

We now have the three difference equations (2.9a) (2.9b) and (2.16) for the three unknowns $x_n$, $\dot{x}_n$ and $\delta_{Pn}$. Their solution will provide the result of our model of the hybrid calculation.

## Solution by z-transform

The z-transform provides a simple method of solving the difference equations. The definition of the z-transform of the sequence $x_n$ is

$$x^* = \sum_{n=0}^{\infty} x_n\,z^{-n} \qquad (2.17)$$

and for our purposes its important property is

$$\sum_{n=0}^{\infty} x_{n+1}\,z^{-n} = z(x^* - x_0) \qquad (2.18)$$

The inversion of a z-transform follows easily by observing from the definition (2.17) that

$$z^{k-1}\,x^* = \sum_{n=0}^{\infty} x_n\,z^{k-n-1}$$

If this is looked upon as a Laurent expansion in the complex variable z the residue is the coefficient of the term for which $n = k$, which is $x_k$. Thus the inversion of $x^*$ to find $x_n$ is accomplished by finding, for each n,

$$\text{Residue}\ (z^{n-1}\,x^*) = x_n \qquad (2.19)$$

The stability of the solution is also indicated by (2.19). Stability requires that $x_n$ not grow as n increases. The only factor in the residue which depends on n is $z^n$, which grows or decreases with n depending on whether the absolute value of z is greater or less than unity. This leads to the well-known stability criterion that every root of the denominator of $x^*$ must have absolute value equal to or less than unity.

The transformation of (2.9) and (2.16) is accomplished by multiplying by $z^{-n}$ and $z^{-n+1}$ respectively, summing and using (2.17) and (2.18), remembering the initial conditions are zero. The result is

$$\begin{aligned}(z - 1)x^* - \dot{x}^*&(e^{-\omega\zeta T}/\omega\zeta^1)\sin\omega\zeta^1T\\ &+ (x^* - \delta_P^*)\,[1 - e^{-\omega\zeta T}\,(\cos\omega\zeta^1T\\ &+ \zeta/\zeta^1\sin\omega\zeta_1T)] = 0\end{aligned} \qquad (2.20a)$$

$$\begin{aligned}\dot{x}^*\,[z - e^{-\omega\zeta T}\,&(\cos\omega\zeta^1T - \zeta/\zeta^1\sin\omega\zeta^1T)]\\ &+ (x^* - \delta_P^*)\,\omega/\zeta^1\,e^{-\omega\zeta T}\sin\omega\zeta_1T = 0\end{aligned} \qquad (2.20b)$$

$$\begin{aligned}(z + K)x^* + \dot{x}^*K\omega^{-1}\,&[\omega\tau + (\ell - 2\zeta h\omega\tau)\omega T\\ &- 2\zeta k\omega^2T^2] - (x^* - \delta_P^*)\,[z + K\,(h\omega\tau\omega T\\ &+ k\omega^2T^2)] = z^2Kx_c/(z - 1)\end{aligned} \qquad (2.20c)$$

These equations have been arranged so the variables are the z-transforms $x^*$, $\dot{x}^*$, and $x^* - \delta_P^*$, and their solution gives the z-transforms of the problem variables, which must then be inverted to yield formulas for the actual solution.

If the three equations are solved by determinants the denominator is given by the determinant of the coefficients,

$$\begin{aligned}\Delta = {}& -z\,[(z-1)^2 - 2z(e^{-\omega\zeta T}\cos\omega\zeta^1 T - 1) \\ & + (e^{-2\omega\zeta T} - 1)] + K\{(z+1)(e^{-\omega\zeta T}\cos\omega\zeta^1 T - 1) \\ & - (e^{-2\omega\zeta T} - 1) + (z-1)1\,\zeta^1 e^{-\omega\zeta T}\sin\omega\zeta^1 T \\ & [\zeta - \omega\tau - (\ell - \zeta h\omega\tau)\omega T + \zeta k\omega^2 T^2] \\ & - (z-1)[(z-1) - (e^{-\omega\zeta T}\cos\omega\zeta_1 T - 1)] \\ & (h\omega\tau\omega T + k\omega^2 T^2)\} \qquad (2.21)\end{aligned}$$

This is a cubic in z, whose roots determine the solution through their residues, according to (2.19).

The solution for x* is then

$$x^* = \frac{z^2 K x_c}{(z-1)\,\Delta}\,[(z+1)(e^{-\omega\zeta T}\cos\omega\zeta^1 T - 1)$$

$$- (e^{-2\omega\zeta T} - 1) + (z-1)\,\zeta/\zeta^1\, e^{-\omega\zeta T}\sin\omega\zeta^1 T] \qquad (2.22)$$

An additional root at z = 1 is visible here, whose residue also makes a contribution.

## Expansion of roots

The nature of the roots of Δ can be seen by letting T approach zero in (2.21). Then all terms approach zero except the first, so one root must approach zero, the other two approach unity. The exact roots are complicated to find since (2.21) is cubic, but we can be satisfied with expansions of the roots in powers of ωT.

Let us first look for a root of the form:

$$z = 1 + d\omega T + e\omega^2 T^2 + f\omega^3 T^3 + \cdots \qquad (2.23)$$

If the coefficients of (2.21) are also expanded in powers of ωT, and (2.23) is inserted, setting the lowest two powers of ωT to zero yields

$$d^2 + (2\zeta + K_T)d + (1+K) = 0 \qquad (2.24)$$

$$e = \tfrac{1}{2}d^2 - \frac{Kd[d\omega_T(h - 3/2) + (\ell - 3/2)]}{2(d + \omega_T\,\zeta_T/\omega)} \qquad (2.25)$$

These determine the first two coefficients in (2.23). The solution of (2.24) is

$$d_{1,2} = \omega_T(-\,\zeta_T \pm i\zeta^1{}_T)/\omega = \lambda_{T1,2}/\omega \qquad (2.26)$$

where $\omega_T$, $\zeta_T$, $\lambda_T$ are defined in (2.3) and (2.4). Thus the first coefficient is identical with the exponent of the exact solution.

To see the significance of this, remember that the important term in the residue is $z^n$ which can be written exp(n ℓn z). But z in the form (2.23) can be used to expand ℓn z to yield

$$z^n = \exp\{nd\omega T + n(e - d^2/2)\omega^2 T^2$$

$$+ n[f - d^3/6 + d(e - d^2/2)]\,\omega^3 T^3 + \cdots\} \qquad (2.27)$$

Thus the first term is part of the exact solution at t = nT, and subsequent terms are error terms.

With two roots $z_1$, $z_2$ given as a complex conjugate pair by (2.23)–(2.26), the third root is simple to find by dividing Δ by $(z - z_1)$ $(z - z_2)$. The expanded result is, using (2.24) and (2.25),

$$\begin{aligned} z_3 = {}& (1-h)\,K\omega_T\,\omega T \\ & + K[\ell - k - 1/2 + \zeta\omega_T(1-2h) \\ & + K(1-h)\,\omega^2\tau^2]\omega^2 T^2 + \cdots \qquad (2.28)\end{aligned}$$

The solution is usually stable to the roots $z_1$, $z_2$ because the real part of d is negative, so the dominant term of $z^n$ is a damping. However, it may be unstable to $z_3$, and will be for large enough ωT.

Before choosing values for the compensation parameters, we will look at the actual solution generated by these roots.

## Solution in the physical (time) domain

The solution is the sum of the residues of $(z^{n-1}\,x^*)$ at the poles $z = 1, z_1, z_2, z_3$, with x* given by (2.22). The residue at z = 1 is easily found by putting z = 1 into (z − 1) x*, which yields

$$\text{Residue}\ (z = 1) = Kx_c/(1+K) \qquad (2.29)$$

which is just the constant part of the exact solution (2.4).

Since $z_1$ and $z_2$ are complex conjugates, so are their residues, and their sum is twice the real part of either. If the expansion (2.23) is put into (2.22) and (2.21), the result for $z_1$ to order ωT is found to be

$$\text{Residue}\ (z_1) = \frac{z_1{}^n\,Kx_c}{2(1+K)}$$

$$\left[-\left(1 + \frac{i\beta_r\omega T}{\omega_T\zeta_T/\omega}\right) + \frac{i\zeta_T}{\zeta_{1T}}\left(1 - \frac{\beta_i\omega T}{\omega_T\zeta_T/\omega}\right)\right] \qquad (2.30)$$

$$e_1 - d_1{}^2/_2 \equiv \beta_r + i\beta_i$$

Finally, the residue at $z_3$ is found similarly using (2.28):

$$\text{Residue } (z_3) = 2^{-1}(\omega T)^{n+3}\,[K\omega_T(1 - h)]^{n+1} \qquad (2.31)$$

## Choice of compensation constants

Comparison of (2.27) and (2.29) with the exact solution (2.4) shows that the first deviation of both the $z^n$ factor, and the rest of the expression, depend on $e - d^2/2$. If this term is zero, the deviation will then be $0(\omega^2T^2)$ in both places. And (2.31) shows that the contribution of the extraneous solution is of high order in $\omega T$ and should decrease rapidly as long as $|z_3| < 1$.

These observations lead to the conclusion that we should make $e - d^2/2$ vanish, which means, according to (2.25),

$$h = \ell = 3/2 \qquad (2.32)$$

The coefficient k is not determined to this order. However, if $e - d^2/2 = 0$ the next term in (2.27) is found from the expansion of (2.21) to be

$$f - d^3/6 = \frac{-Kd\{d[k - 13(1 + d\omega_T)/12] - 2K\omega_T(1 + d\omega_T)/3\}}{2(d + \omega_T\zeta_T/\zeta)}$$

This cannot vanish for any choice of real k. One can make either its real part or its imaginary part vanish, although k will then depend on the parameters of the problem. One obvious choice which reduces the size of $f - d^3/6$ is

$$k = 13/12 \qquad (2.33)$$

and this is the one used in the implementation. Further study would be needed to determine if another, more complicated, choice were better.

Notice that the values given in (2.32) are exactly those shown in (2.15), which are obtained by projecting to $(n + 1/2)T$, while the k of (2.33) is only 1/24 smaller than the corresponding value of k in (2.15). One can therefore look upon the analysis as providing a derivation of the length of the projection interval, in contrast with the usual graphical or intuitive arguments.

## Results for three-step compensation

An entirely analogous solution can be obtained using the three-step projection of (2.13). The necessary starting values $\delta_{-1}$ and $\delta_{-2}$ are taken the same as $\delta_0$.

The determinant of the coefficients is now fifth degree, with five roots. Two are of the form (2.23) with d the same, (2.24). The next coefficients are

$$e - \tfrac{1}{2}d^2 = \frac{Kd(1 + d\omega_T)(a_1 + 2a_2 + 3/2)}{2(d + \omega_T\zeta_T/\zeta)} \qquad (2.34)$$

and, if $e - d^2/2 = 0$,

$$f - d^3/6 = \frac{-K(1 + d\omega_T)[Kd\omega_T/12 + d^2(a_2 - 22/12)]}{2(d + \omega_T\zeta_T/\zeta)} \qquad (2.35)$$

The other three roots are power series in $(\omega T)^{1/3}$, given in terms of

$$r = (-1 + i3^{1/2})/2, \quad \bar{r} = (-1 - i3^{1/2})/2$$

by

$$z_{3,4,5} = (K\omega_T a_2\omega T)^{1/3}(1, , \bar{r}) \qquad (2.36)$$
$$+ (K\omega_T a_2\omega T)^{2/3}(a_1 + a_2)(1, \bar{r}, r)/3a_2$$
$$+ K\omega_T a_2\omega T/3a_2 + \cdots$$

The residues at $z = 1$ and $z = z_1$ are the same as for Taylor series compensation, (2.29) and (2.30). The first terms of the residues of the other three roots are

$$\text{Residue } (z_{3,4,5}) = (Kx_c/6)(\omega T)^{(n+7)/3}$$
$$(K\omega_T a_2)^{(n+1)/3}\,(1, r, \bar{r})^{n+1} \qquad (2.37)$$

To make the $0(\omega T)$ errors vanish we make $e - d^2/2 = 0$ by taking

$$a_1 + 2a_2 = -3/2$$

which agrees with (2.14). To determine $a_1$, $a_2$ separately one can go to (2.35) and choose $a_2 = 22/12$, which is 1/24 less than the value in (2.14). So again we come very close to the usual projection distance by an analytical derivation.

The error caused by the extraneous roots should not be as small for this type of compensation, since it depends on $(\omega T)^{n/3}$, and decreases rather slowly, as n increases.

The solution is also less stable, because of the one-third power dependence of the roots on $\omega T$. In fact, the absolute values through the first two terms are

$$|z_{3,4,5}| = (K\omega_T a_2 \omega T)^{1/3}$$
$$|1 + (1, -\tfrac{1}{2}, -\tfrac{1}{2})(K\omega_T a_2 \omega T)^{1/3} (a_1 + a_2)/3a_2| \quad (2.38)$$

and since $a_1 + a_2$ is negative, the conjugate pair $z_4$, $z_5$ is the least stable. This is the pair introduced by going back two steps in time, which shows the destabilizing influence of that procedure.

## Stability considerations

As mentioned already, it is the extraneous roots which control the stability of the hybrid calculation. For the Taylor series compensation, this root is given by (2.28), and is of the order $K\omega_T\omega T$, the same as for the uncompensated case, which can be obtained from (2.28) by putting $k = h = \ell = 0$. In fact, the compensated root is somewhat smaller (thus more stable) since the coefficient of the first term is $-1/2$ instead of 1. Notice that one could improve the stability, at some cost in accuracy, by choosing k so that the coefficient of the second term in $z_3$ vanishes, although k would then depend on the parameters of the problem instead of being constant.

In contrast, the extraneous roots for three-step compensation are given in (2.38) and are of order $(a_2\ K\omega_T\omega T)^{1/3}$, considerably larger than the uncompensated or Taylor series cases. Therefore, the three-step method yields a less stable solution. If $a_2 = 0$, we then have a two-step scheme, and there are only two extraneous roots, of order $(K\omega_T\omega T)^{1/2}$, more stable then the three-step scheme but still less stable than the uncompensated or Taylor series cases.

If the scheme of Howe and Fogarty, discussed in the Introduction, were used, there would also be two extraneous roots of order $(K\omega_T\omega T)^{1/2}$, so the stability would be about the same as for a two-step scheme. In fact, the two-step and Howe-Fogarty schemes are closely related, both going back one step in time.

*Computer implementation*

The Taylor series (or derivative) method of compensation was tested, and compared with the three-step method, by solving the problem posed by (2.1), (2.2) on the hybrid computer of the NASA Electronics Research Center. This is a Beckman 2200/SDS 9300 machine with interface built by Beckman.

As described above, integration the of x and x, and the combination of x and x on the left side of (2.1) were performed in the analog computer. The value of $\delta$ was found in the digital computer, by sampling x and x from the analog at intervals of T and extrapolating. Then $\delta_P$ was calculated and sent back to the analog to be used to find x. The A to D sampling was accomplished first, followed immediately by the D to A updating. In order to compare the resulting hybrid solution with a continuous solution, the complete equation was also solved in the analog simultaneously as an oscillator with frequency $\omega_T$ and damping $\zeta_T$ as defined by (2.3). The details of the analog circuit, the digital programs, the control circuit, the scaling, etc., are given in Ref. 7, pages 130-141 and Appendix E.

The output of this calculation was a set of strip-charts and digital printouts giving the hybrid and pure analog values of x, $\dot{x}$, x, $\delta$, and the difference between the hybrid and analog values, which may be taken as a measure of the error of the hybrid solution.

Runs were made for the parameters

$$\omega = 0.412,\ \zeta = -0.2425,\ \zeta_T = 0.7$$

using the conventional compensation constants

$$a_1 = -21/4,\ a_2 = 15/8,\ a_0 = 1 - a_1 - a_2 = 35/8$$

for the three-step method, and the set

$$\ell = h = 3/2,\ k = 13/12$$

which we have derived for the Taylor series method. The values of $\omega_T$ were varied between 0.5 and 15.0. For each such value, the control parameters K and $\tau$ can be calculated from (2.3). Runs were made at several sample intervals T in order to study the stability of the hybrid calculation. For large enough T is was always possible to make it unstable.

The relative merits of the Taylor series and three-step compensation schemes, compared to pure analog and uncompensated hybrid results, are strikingly illustrated by excerpts from the strip charts drawn by the analog computer. The case chosen for illustration is $\omega_T = 15$, for which (2.3 gives $K \doteq 1234$, $\tau = 0.0942$.

Figure 1 shows the strip chart record for x(t) for four cases. At the top is the continuous solution produced by a pure analog calculation. Below follow the records for the uncompensated, Taylor series compensated, and three-step compensated hybrid solutions all for a sample interval T = 25 milliseconds, which is 17 samples per cycle based on total frequency. In order to bring out the errors more clearly, Figure 2 shows the difference signal $x_H - x_A$ on a larger scale, where the subscripts H and A stand for hybrid and analog, re-

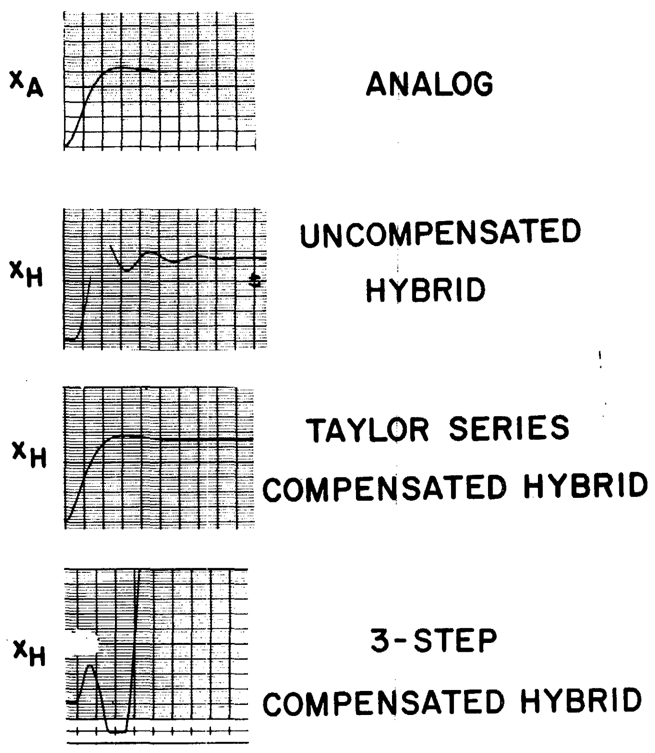


Figure 1—Strip chart records of x(t) for $\omega = 0.412$, $\zeta = -0.2425$, $\zeta_T = 0.7$, $\omega_T = 15$. The sample interval T = 25 ms.

spectively. The great improvement in accuracy achieved by going from no compensation to Taylor series to compensation is apparent. On the other hand, the solution with three-step compensation is unstable and saturates the amplifiers.

The stability properties of these three cases are predicted by the formulas we have developed. For no compensation ($\ell = h = k = 0$), (2.28) gives $|z_3| = 0.736$, while for Taylor series compensation the same formula shows $|z_3| = 0.411$. On the other hand, for three-step compensation, (2.38) gives $|z_3| = 0.578$, $|z_4, z_5| = 1.375$. Therefore, the part of the solution corresponding to the root $z_3$ is stable, but the part corresponding to the roots $z_4$, $z_5$ are unstable, leading to an unstable solution, as shown in Figures 1 and 2. To stabilize the three-step case, the sample interval T would have to be reduced to 10 ms, or about 42 samples per cycle, for which (2.38) shows $|z_4| = 0.928$. A case run at this value of T indeed showed three-step compensation to yield a stable solution.

To destabilize the uncompensated and Taylor series cases, a run was made at T = 50 ms (8.4 samples per cycle), for which (2.28) gives $|z_3| = 1.89$ and 1.12, respectively. The results of the run are shown in Figure 3, where the rapid increase of x until the amplifiers saturate is seen for both cases.

Similar results hold for other values of $\omega_T$. In all cases, stability or instability exhibited by the numerical calculations could be predicted in advance by use of (2.28) or (2.38). Furthermore, the digital printouts showed that with the same set of parameters and sampling interval, the Taylor series method gave more accurate results, that is, results closer to the analog (continuous) solution. The improvement in accuracy

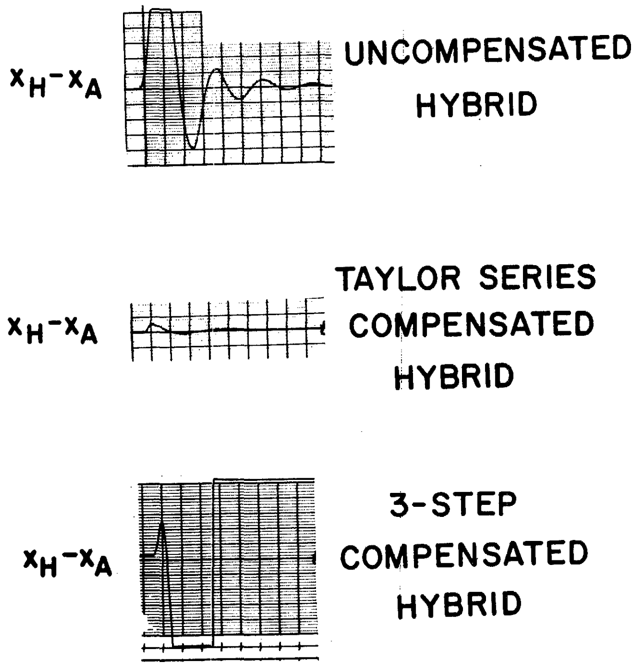


Figure 2—Strip chart records of $x_H$ (HYBRID) $-x_A$ (ANALOG) for $\omega = -0.412$ $\zeta = 0.2425$, $\zeta_T = 0.7$, $\omega_T = 15$. The sample interval T = 25 ms.

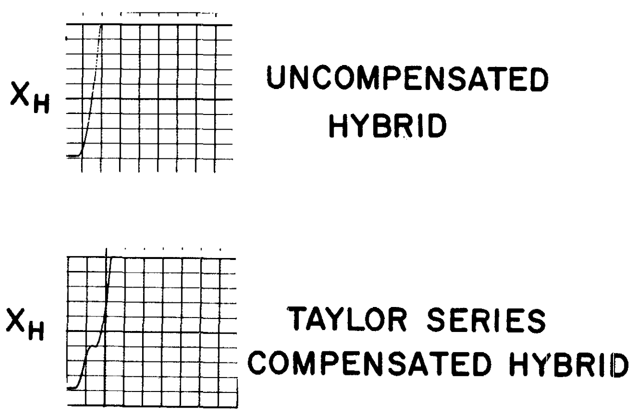


Figure 3—Strip chart records of x(t) for $\omega = 0.412$ $\zeta = -0.2425$, $\zeta_T = 0.7$, $\omega_T = 15$. The sample interval T = 50 ms.

could be quite marked for sample intervals near the stability limit of the three-step method. This is in accord with the deductions from the extraneous solutions (2.31) and (2.37).

## CONCLUSIONS

The Taylor series (or derivative) method of compensation appears to have a number of advantages over the three-step method of compensation for the time delay and D to A hold errors of hybrid computing. For a given case, it can be made stable for larger sampling intervals than the three-step method, and is more accurate at the same sampling interval. The Taylor series method can also be made stable for larger sampling intervals than the uncompensated case for almost all values of the parameters, while the three-step method may well be unstable when the uncompensated calculation is stable. In other words, compensating by Taylor series can improve the stability, while compensating by the three-step method destabilizes.

These stability advantages of the Taylor series method depend to a large extent on the particular form of implementation used. The crucial point is not to use information which goes back in time, since each such backward time step introduces an extraneous solution which is de-stabilizing. The implementation suggested here, where the extrapolations are accomplished in the digital computer, avoids extra backward time steps while still permitting the A to D sampling to be done before the D to A transfer. If the extrapolations are done in the analog computer, as in the Howe-Fogarty[5,6] implementation, the A to D before D to A sequence of operations introduces one backward time step and adversely effects the stability. If the sequence is performed in the order D to A followed by A to D, the analog extrapolation of Howe and Fogarty would give exactly the results of the present analysis.

The z-transform method of analysis for linear equations can be carried through with arbitrary coefficients in the extrapolation formulas. Then they can be chosen to yield the desired improvement in accuracy and/or stability. The coefficients of the first power of the sample interval T clearly should be chosen to extrapolate by 1.5T, but the coefficient of $T^2$ in the extrapolation formula for x has some flexibility in the choice, depending on whether accuracy or stability is the paramount consideration.

When the derivatives are available, there is no more difficulty implement the Taylor series method than the three-step method, and there are no starting problems with the former, as there are with the latter.

On the basis of the analysis and numerical results of this study, the Taylor series method of compensation seems preferable in all ways to the three-step method, and can be recommended whenever the derivatives are available. Whether this conclusion also will hold for non-linear equations and for higher order systems, depends on the results of applying the Taylor series method to those cases. Some preliminary study of a linear fourth order system by the present method of analysis indicates that the Taylor series method may be applicable, but with different values of the compensation coefficients.

## ACKNOWLEDGMENTS

This work was performed under Contract No. NAS-12-676 with NASA Electronics Research Center, Cambridge, Massachusetts 02139. The collaboration of Joseph Krawiec and Albert Shortell in programming and running the analog computer is gratefully acknowledged.

## REFERENCES

1 E E L MITCHELL
*The effect of digital compensation for computation delay in a hybrid loop*
Proc FJCC Vol 31 1967 103-108

2 T MIURA J IWATA
*Effects of digital execution time in a hybrid computer*
Proc FJCC Vol 24 1963 251-265

3 E G GILBERT
*Dynamic error analysis of digital and combined analog-digital computer systems*
Simulation Vol 6 1966 241-257

4 W J KARPLUS
*Error analysis of hybrid computer systems*
Simulation Vol 6 1966 120-136 Reprinted in Mc Leod J Ed Simulation McGraw-Hill Book Co Inc N Y 1968 65-80

5 G A BEKEY W J KARPLUS
*Hybrid computation*
John Wiley and Sons Inc N Y 1968 Chapt 5

6 R M HOWE L E FOGARTY
*Error analysis of computer mechanization of airframe dynamics*

7 Wolf Research and Development Corporation
Final Rpt on Contract No 12-676 NASA Electronics Research Center Cambridge Mass Dec 1968

# HYPAC—A hybrid-computer circuit simulation program

*by* PHILIP BALABAN

*Bell Telephone Laboratories*
Holmdel, New Jersey

## INTRODUCTION

Computer simulation of electronic circuits and systems has become an increasingly important tool in circuit and system design. Such simulations enable one to:

1. Eliminate the necessity of building many breadboard models in order to evaluate different design approaches.
2. Analyze the performance of the circuit as a function of different parameters.
3. Model semiconductor devices and integrated circuits so that intrinsic parameters become accessible.
4. Perform optimization and tolerance analysis of a circuit which requires many evaluations of the circuit with different sets of parameters.

The frequency domain analysis of a linear circuit is usually simulated on the digital computer and the solutions obtained are accurate and fast. The analysis in the time domain is more difficult with both analog and digital computers often being used for this purpose. The analog computer simulation technique features fast solution times and designer interaction; the digital computer simulation programs have the advantage of a large dynamic range and very simple programming.

Unfortunately, both types of simulations can handle only relatively small circuits, since the analog computer is hardware limited (a six to eight transistor circuit can be patched on a large analog computer) and the digital computer requires an excessive amount of computation time, especially when the eigenvalues of the system are far apart.

The HYPAC hybrid computer program was developed in order to overcome the above-mentioned shortcomings for a special class of problems. This class includes systems that have a modular structure where a few types of a particular subcircuit (amplifier, gate, etc.) are used repetitively. Such systems are very common, especially since the advent of integrated circuits.

### *The program structure*

The program takes advantage of the speed of the analog computer and the possibility of storing information on the digital computer. The principle is the following:

A whole subcircuit (such as an amplifier or gate) is patched on the analog computer and multiplexed by the digital computer to form a large system. Thus, the digital computer regards this one subcircuit as N distinct subcircuits. Each subcircuit has its own distinct inputs and parameters which the digital computer provides sequentially to the analog computer. In addition to the interconnection and memory capability, it is very convienent for the multiplexer to have the capacity to model different circuit elements. Therefore, a general purpose, widely used block-oriented digital program called "PACTOLUS"[1] was chosen as the vehicle for this hybrid program. Hence, the name *HY*brid *PAC*TOLUS—HYPAC.

### *Description of PACTOLUS*

PACTOLUS can be described as a block-oriented interpretive language. The program incorporates all standard analog computer elements (integrators, summers, multipliers, etc.). In addition, the program allows a few special elements of unspecified function. The user may write his own subroutine for any function he desires.

We shall consider the computing operation procedure implied by Figure 1 where

$X_n$ = the input vector at $t = n\Delta T$

$Y_n$ = the output vector at $t = n\Delta T$

$\dot{Y}_n$ = the derivative vector at $t = n\Delta T$

At time $t = 0$, all the input vectors $[X_0]$ and the state vector $[Y_0]$ (the initial conditions of all integrators) are given. From these given conditions, the derivative vector $[\dot{Y}_n] = F([X_n, Y_n])$ can be computed. $[Y_{n+1}]$ is computed from $[\dot{Y}_n]$ using any integration method. In the original PACTOLUS, a second order Runge-Kutta method is used.

### *Modification of PACTOLUS for HYPAC*

The original PACTOLUS was supplemented and modified in many ways to convert it into a general purpose hybrid program. Some of the added features are briefly illustrated below.

#### The hybrid element

The hybrid element was conceived to be a special element of the PACTOLUS repertoire. The analog subcircuit of the hybrid element is patched on the analog computer as in Figure 2.

The inputs and the initial conditions are supplied through D/A converters by the digital computer. The outputs of the circuit and the outputs of all the integrators are fed back to the digital computer through A/D converters.

In HYPAC, the hybrid block is sorted as an integrator, which implies that at time $t = 0$, the outputs of the circuit and the initial conditions of the integrators are known. The inputs to the circuit are then evaluated from the given values and applied through the D/A converters to the inputs of the analog block. The stored initial conditions (IC) for each integrator are applied to the appropriate integrators. After the integrators have settled to their respective initial conditions, the analog computer is switched into the operate mode for a time $\Delta T$.* At the end of this period, the outputs of this circuit are sampled and transferred to the digital computer. The outputs of the integrators are also sampled and stored in the digital computer to be used as initial conditions in the next time period. If N such circuits are used in the system, the HYPAC program regards them as N distinct subcircuits. The inputs and outputs are regarded as coming from distinct blocks and used accordingly.

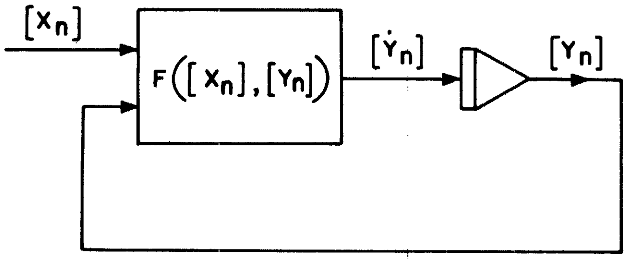


Figure 1—Principle of operation of PACTOLUS

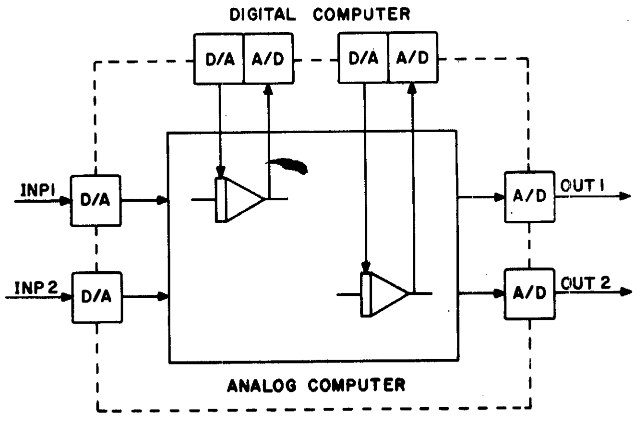


Figure 2—The hybrid element

The subcircuits must have an identical topology, the parameters, however, can be different and can be changed before each run using digitally controlled attenuators.

#### The output configuration

The original PACTOLUS allows only one output per element. Since the hybrid element is actually a complex circuit, it will usually have more than one output. Therefore, another special element "$\phi$" was assigned to handle multiple outputs. For each additional output, one such fictitious output element "$\phi$" is assigned. In Figure 3, the second and third outputs of the hybrid element No. 1 are read out from output elements 1 and 3.

The output elements simply store the value of the output A/D converters. The D/A and A/D converters

* In order to operate reasonably efficiently, and accurately, the analog computer must have electronic mode control.

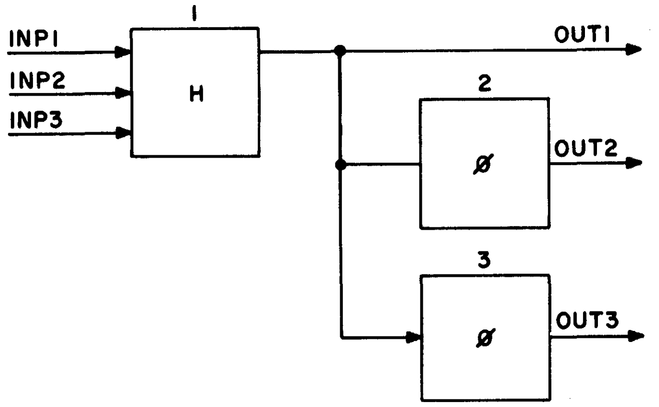


Figure 3—The output configuration of the hybrid element

usually come in pairs, therefore, this program is coded so that to each input corresponds an output, i.e., to input No. 1 corresponds output No. 1; to input No. 2, output No. 2. This associates each output with a corresponding D/A converter.

## The integration algorithm

The integration algorithm was changed to the simple Euler method since this is closest to the way an analog integrator performs in the hybrid element. However, a closed loop integration method and a variable integration step method are now being studied.

### *Detailed operation of the hybrid program*

As noted before, the hybrid element is sorted as an integrator. All inputs $[Y_n]$ to the hybrid elements and inputs $[\dot{I}_n]$ to the integrators at time $t = n\Delta T$ (shown in vector form in Figure 4) are computed from the output vectors $[Z_n]$ and $[I_n]$ of these elements and the input vector $[X_n]$ to the circuit at time $t = n\Delta T$. The computation of $[I_{n+1}]$ is described in an earlier section.

The computation of $[Z_{n+1}]$ from $[Y_n]$ and the corresponding initial conditions can be understood by examining the sequence of operations shown in the timing diagram in Figure 5.

All the commands and timing signals of the analog computer are generated in the digital computer. First, the analog computer is switched into the "IC" (initial condition) mode (1) in Figure 5. Then the parameters of the hybrid element are adjusted through the digitally controlled attenuators (DCAs) (2). The inputs and the corresponding initial condition voltages are then applied to the hybrid element (3). The hybrid element is left in this mode for a constant time necessary for the integrators to settle. The analog computer is then

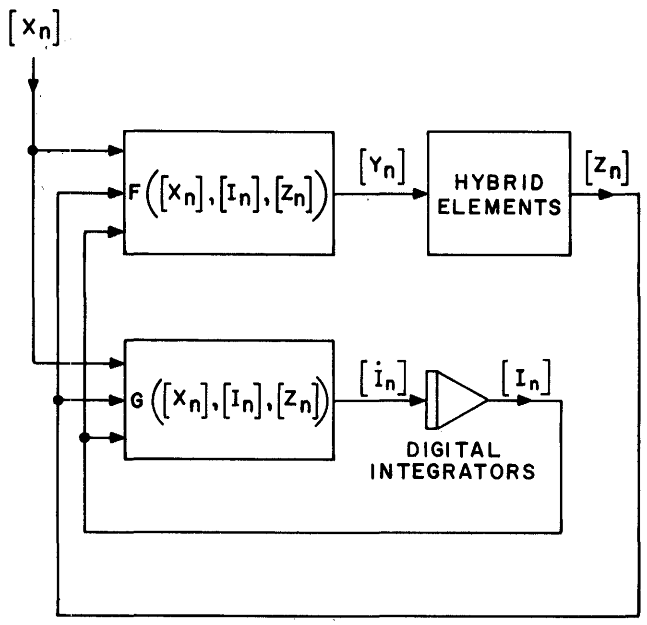
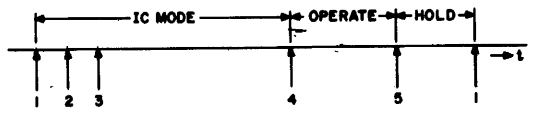


Figure 4—Principle of operation of HYPAC



SEQUENCE:

1. ANALOG COMPUTER SWITCHED INTO "IC" MODE.
2. DIGITALLY CONTROLLED PARAMETERS SET.
3. INPUTS AND INITIAL CONDITIONS APPLIED.
4. ANALOG COMPUTER SWITCHED INTO "OPERATE" MODE.
5. ANALOG COMPUTER SWITCHED INTO "HOLD" MODE AND $Z_n$ IS READ INTO DIGITAL COMPUTER.

Figure 5—Sequence of operation of the hybrid element

switched into the "OPERATE" mode for the time $\Delta T$ which is specified by the user at the beginning of each run. At the end of this period, the analog computer is switched into the "HOLD" mode for a very short time (100usec) long enough for the A/D converters to be read out. The outputs of the A/D converters are then the outputs of this hybrid element at time $t = (n + 1)\Delta T$. The analog computer is ready to be used as the next hybrid element.

## Selection of the integration interval $\Delta T$

The overall circuit has to be scaled in the time domain to analog computer compatible frequencies, usually smaller than 1 kHz. The smallest possible $\Delta T$ which is provided in the program is one msec. In order for the solution to converge, the eigenvalues of the circuit should not exceed the value of $2/\Delta T =$

$2 \times 10^3$.** (This restriction applies only for hybrid and digital loops in the circuit and *not* to loops confined to the analog computer.) Therefore, selection of the time scale defines the upper bound of the integration interval. On the other hand, $\Delta T$ should be large enough in order to minimize the effect of noise and truncation by the A/D converter. It is, therefore, very important to scale the analog computer so that the integrators work at the highest possible level of the voltage range.

### Selection of the initial condition settling time

The initial condition settling time is dependent on the value of the integrating capacitor and the maximum current of the output stage of the operational amplifier and is, therefore, dependent on the computer used. However, in any analog computer, the IC settling time is directly proportional to the integrator capacitance. Therefore, the overall computing time is substantially reduced by choosing a faster integrator mode. On the other hand, the impairments introduced into the integrators by switching transients and integrator drifts are inversely proportional to the value of the integrator capacity.

The experimental example described in a later section was run both with $1\mu F$ capacitors with 20 msec IC settling time and $.01\mu F$ capacity with 1 msec IC settling time. No noticeable deterioration of accuracy was detected in this example, but this may not be generally true and if solution time is not critical, a larger integrator capacitor should be used.

### Accuracy considerations

HYPAC is both a digital and analog system, therefore, all factors that produce errors in digital and analog differential analyzers will also produce error in this system. These factors are many and are extensively covered in literature.[2,3,4] They include finite sampling, round off and quantization in the digital system and limited bandwidth, noise (limited dynamic range), accuracy and linearity of components, etc., in the analog system.

The accuracy consideration which is unique to this program is connected to the way the initial conditions are set up in the hybrid element.

1. The initial condition of any particular run has come through an Analog to Digital to Analog conversion string and was therefore truncated by the A/D converter.
2. Switching transients are generated whenever the integrators are switched from one operation mode to another. These transients are caused by charges stored in the parasitic capacities of the switches, and affect the outputs of the integrators. The magnitude and polarity of these small voltage increments caused by switching transients can be regarded as random.[5]

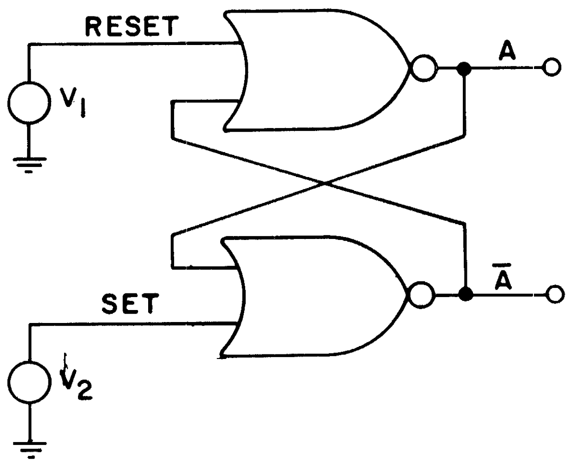
The error contributions of these two factors are minute for each integration period $\Delta T$ (a few millivolts) but since a solution usually consists of a few hundred integration periods, the propagation of these errors can be very significant. It is the author's feeling that the errors will not build up if the system itself and the hybrid element in particular is stable. Although our experiments have confirmed this, the above statement is rather intuitive and needs further investigation.

*Programming of HYPAC*

In order to demonstrate how a problem is prepared for simulation, let us consider a simple example.

The circuit in Figure 6a is a set-reset flip-flop composed out of two NOR gates, a positive voltage $V_1$ resets the flip-flop and $V_2$ is the set input. The circuit diagram of the NOR gate used is shown in Figure 6b.

The complete circuit is depicted in Figure 7. Let us assume that it was decided to use the outlined subcircuits as the hybrid element in this example. Since this element is not buffered from its input and output circuits, some special approach is needed in



a

Figure 6a—Set-reset flip-flop

** If the circuit is nonlinear, the eigenvalues should be evaluated at the worst possible combination of parameters and biases.

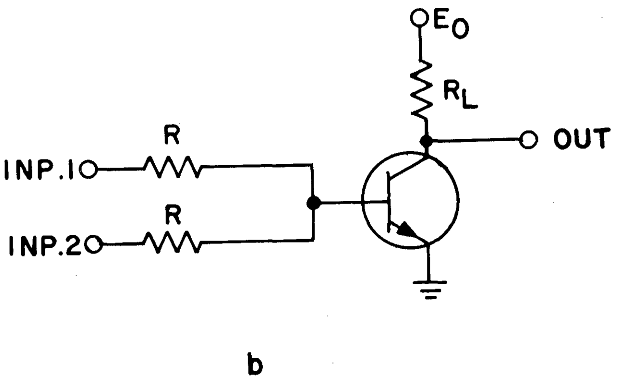


Figure 6b—NOR gate

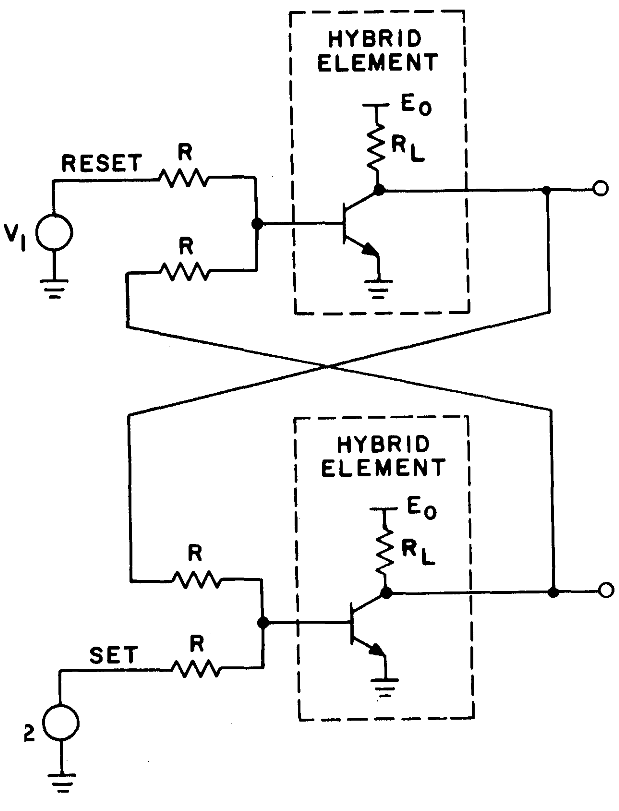


Figure 7—Set-reset flip-flop

order to extract this subcircuit from the whole circuit. The approach we used is called the partition method and is described in the Appendix.

Using this method, the circuit can be partitioned into blocks as shown in Figure 8.

This circuit is identical to the one in Figure 7 if the relations

$$I_{B1} = i_{B1};\ V_{B1} = -v_{B1};\ I_{O1} = i_{O1};\ V_{O1} = -v_{O1}$$

$$i_{B2} = i_{B2};\ V_{B2} = -v_{B2};\ I_{O2} = i_{O2};\ V_{O2} = -v_{O2}$$

are satisfied.

The programming of HYPAC is reduced to writing the nodal or loop equations of the circuit. The HYPAC block diagram is shown in Figure 9. The blocks H1 and H2 are hybrid elements, blocks $\phi 1$ and $\phi 2$ are associated with the second outputs of the hybrid element, all other blocks are conventional PACTOLUS elements.

The hybrid element can be patched on the analog computer as shown in Figure 10. The transistor model used in this simulation is the analog separation model[6] based on the charge control equations. The input $I_B$ and $I_0$ and the initial conditions are applied through D/A converters. The outputs of the subcircuit and the outputs of each integrator are read out through A/D converters. The outputs of the integrators are stored to be used as the next initial conditions.

While using this method of programming, one should be careful not to introduce hybrid algebraic loops, which, of course, can be highly unstable. Such loops can be easily spotted by inspection and can usual-

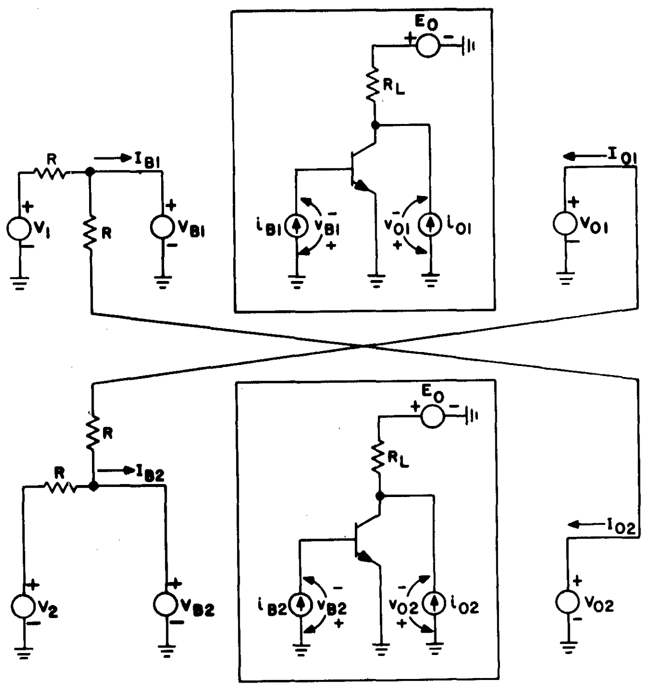


Figure 8—Set-reset flip-flop

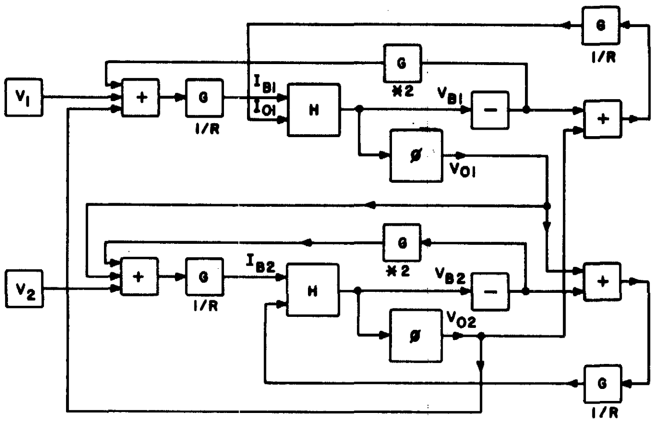


Figure 9—HYPAC block diagram for set-reset flip-flop

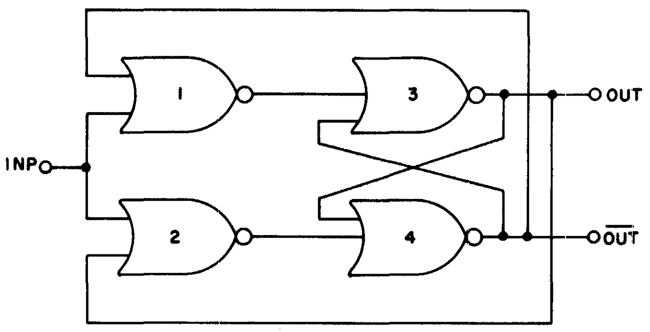


Figure 11—Trigger flip-flop

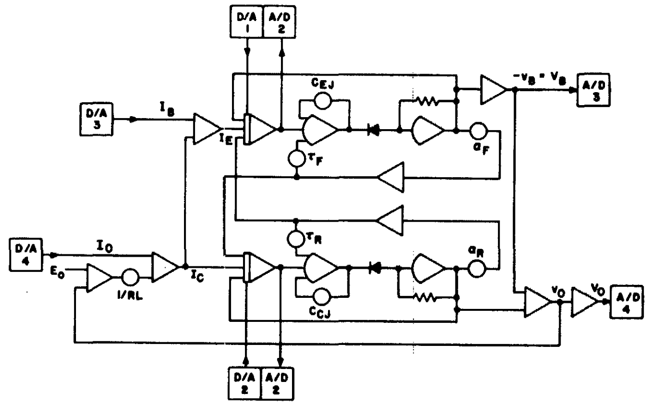


Figure 10—Analog simulation of a NOR gate

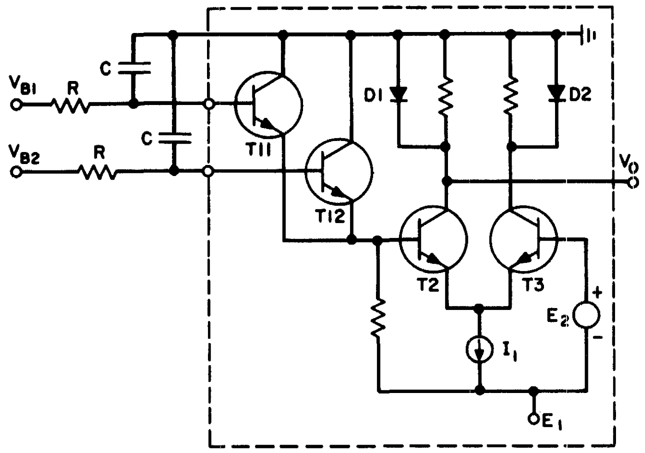


Figure 12—NOR gate

ly be eliminated by placing them wholly in the hybrid element.

*Experimental results*

In order to demonstrate the effectiveness of the program, a more extensive example was programmed.* The example consisted of a trigger flip-flop composed of four half-nanosecond NOR gates. (Figure 11).

The flip-flop changes state every time a negative (zero) pulse is fed to the input. Since the NOR gates are identical, an entire gate was programmed as a hybrid element. The schematic of the half nanosecond NOR gate is depicted in Figure 12. The outlined part was simulated on the analog computer as shown in Figure 13.

* The Hybrid Computer, used in this experiment, consisted of an EAI-8800 Analog Computer linked with a CDC-3300 Digital Computer

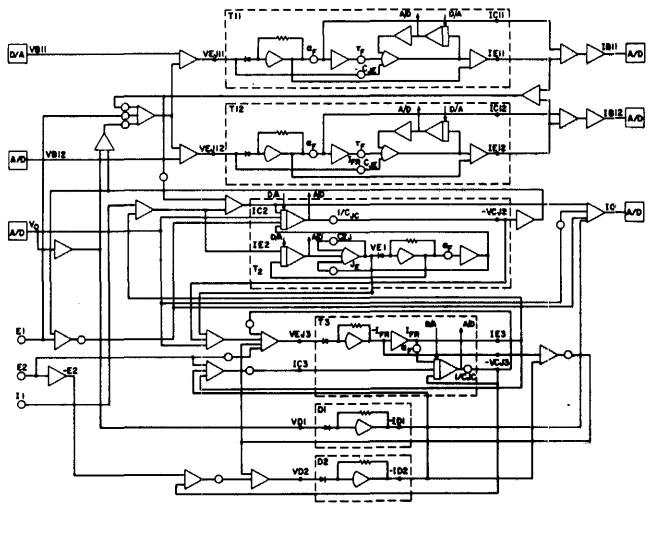


Figure 13—Analog computer simulation of a NOR gate

The transistors and diodes are simulated using the separation technique. Only the relevant reactances for the operation of this gate in the flip-flop circuit were simulated. The HYPAC block diagram of the flip-flop is shown in Figure 14. The R and C elements at the inputs of the gate were included in the digital simulation of HYPAC in order to reduce the number of D/A and A/D converters necessary for simulation.

Since the risetime of the circuit is approximately $t_r = 0.5$ nsec, we assumed that the largest eigenvalues of the circuit must be around $\omega_{\max} = \frac{2}{0.2} t_r = 2 \times 10.^{10}$ To be on the safe side, we chose a time scale $\alpha_T = 10^8$ so that 1 msec machine time = 0.01 nsec real time. The voltage scale was $\alpha_v = 10$ and the current scale $\alpha_I = 1{,}000$.

A typical solution is shown in Figure 15. A is the waveform applied to the input of the circuit (HYPAC element No. 2 in Figure 14), B is the output waveform of the NOR gate No. 1 in Figure 11, or HYPAC element No. 14 in Figure 14. C is the waveform at the output of NOR gate No. 3 in Figure 11 or HYPAC element No. 24 in Figure 14.

The integration interval for this experiment was $\Delta T = 4$ msec corresponding to .04 nsec in real time. The waveforms in Figure 15 represent 16 nsec of the solution time. These waveforms correspond quite closely to ones observed in breadboard experiments.

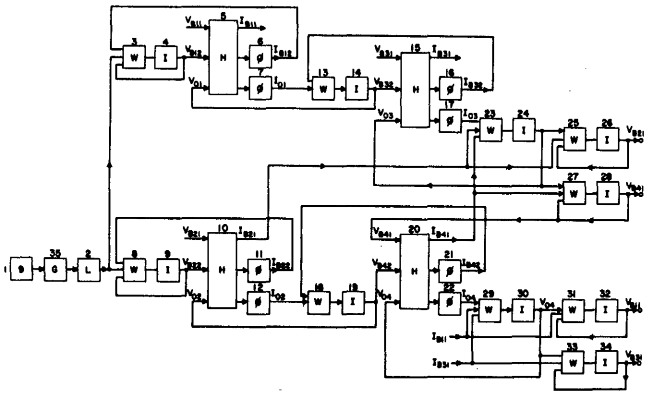


Figure 14—HYPAC diagram of a trigger flip-flop

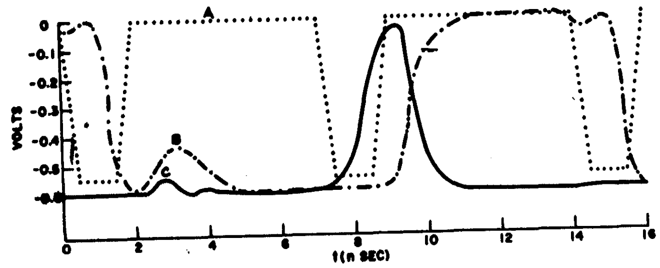


Figure 15—Waveforms of simulated flip-flop

The total computing time was 70 sec for the program when the hybrid element used 20 msec IC settling timep er run, corresponding to 1$\mu$F integrator capacitor value. The computing time was reduced to 40 sec when the IC settling time was 1 msec corresponding to 0.01 $\mu$F integrator capacitor value.

A simplified problem using two half-nanosecond NOR gates was run on the digital computer. The program used was a general purpose circuit analysis program using the state space approach. The resultant waveforms had the same general shape as the one obtained on HYPAC although exact evaluation of the errors was not possible since the circuits were not identical. The computing time for a 16 nsec solution on the digital computer was one hour.

### *Improvements considered*

One of the main difficulties in using this program is the selection of the appropriate integration interval $\Delta T$. In order to simplify this task two integration algorithms will be incorporated into the HYPAC program. The user will have the choice of selection of the appropriate algorithm for his problem.

### Integration algorithm for stiff differential equations

As noted earlier the Euler integration method is stable only if $\Delta T < 2/a_i$ where $a_i$ is the largest eigenvalue of the system. For systems where the frequencies of interest are a few orders of magnitude smaller than the largest eigenvalue the use of this integration method will result in extremely long computational runs. As an example of such a system we can consider a 1 kHz oscillator using "100 MHz" transistors.

A backward integration method where the stability is independent of the integration interval was recently proposed by Sandberg and Shichman.[7] This algorithm uses a Newton-Raphson iteration technique to solve the implicit algebraic equation resultant from the backward integration method. Preliminary investigations suggest that a simiilar integration method can be adopted for HYPAC.

Let $\overline{Z}_n$ denote the output vector of all the Hybrid elements, digital integrators and other time dependent elements at time $t = n\,\Delta T$

then $$\overline{Z}_n = f(\overline{Y}_n, \overline{X}_n, \Delta T) \tag{1}$$

where: $\overline{Y}_n$ is the input vector to the hybrid elements and digital integrators.

$\overline{X}_n$ is the initial condition vector of all the integrators.

The input vector in turn is computed from the algebraic equation

$$\overline{Y}_n = g(\overline{Z}_n, \overline{U}_n) \qquad (2)$$

where $\overline{U}_n$ is the input vector to the circuit.
Combining equations (1) and (2) we get

$$Z_n = f(g(\overline{Z}_n, \overline{U}_n), \overline{X}_n, \Delta T) \qquad (3)$$

This is an implicit equation which can be solved for $Z_n$ using the Newton-Raphson method. For the kth iteration we get

$$\overline{F}_{n,k} = \overline{Z}_{n,k} - f(g(\overline{Z}_{n,k}, \overline{U}_m), \overline{X}_n, \Delta T),\ k = 0 \rightarrow K \quad (4)$$

and

$$\overline{Z}_{n,0} = f(g(\overline{Z}_{n-1},; \overline{U}_n), \overline{X}_n, \Delta T) \qquad (5)$$

The equation is considered solved when $||F_k|| < \epsilon$ where $||F_k||$ is the usual Euclidian norm and $\epsilon$ a preset error criterion.

The Jacobian will be computed by perturbing each output $Z_{n,i}$ by $\Delta Z_{n,i}$ and computing all the partial derivatives $\partial F_{nj}/\partial Z_{n,i}$.

An additional benefit of this method is that hybrid algebraic loops converge to a stable solution.

### Integration algorithm with adaptive integration interval

In order to increase the accuracy of the Euler intergration method, the error produced at the output at each intergrator for every intergration step will be monitored. The integration interval is then adjusted so that this error remains within prescribed bounds.

The Euler intergration method with a variable integration interval is given by

$$Y_{n+1} = Y_n + \Delta T_n Y_n$$

The error E is approximated by the second difference of $Y_{n+1}$ (second term of the Taylor expansion).

$$E_{n+1} = \frac{Y_{n+1} - Y_n\left(1 + \frac{\Delta T_n}{\Delta T_{n-1}}\right) + \frac{\Delta T_n}{\Delta T_{n-1}} Y_{n-1}}{2}$$

Two error levels will be predetermined $E_{min}$ and $E_{max}$. Each integration interval will be adjusted through an iterative routine so that all integration errors will be smaller than $E_{max}$ and at least one larger than $E_{min}$.

Such an adaptive integration method will keep the errors within stable bounds and will reduce the errors caused by truncation of the A/D converters.

## CONCLUSION

A hybrid program has been developed which makes it possible to analyze a special class of large circuits considered untractable by conventional methods. The topology of the circuit has to be modular and composed of identical subscripts.

Effective use of the program requires a degree of sophistication since the programmer has to be familiar with all the intricacies of both analog and digital simulation such as scaling and selection of the integration interval. Although the setup of the problem is relatively time consuming, the reduction of computation time compared to a wholly digital computation solution is dramatic. This reduction of time makes it possible to perform optimization and tolerance analysis.

The program has its most significant value in design of integrated circuits where modular topology is the standard design philosophy.

## ACKNOWLEDGMENTS

The valuable discussions of this material with J. Chernak are gratefully acknowledged. The author is particularly indebted to J. M. Schilling who wrote the HYPAC program used in this work.

## APPENDIX

*Partition of circuit for simulation purposes*

Simulation of circuits on the analog computer generally does not preserve the topology of the circuit. Since every node or branch is described by two variables, voltage and current, these variables are handled separately on the computer and do not appear at the same place (Figure 16).

Therefore, if a circuit has to be partitioned into two (or more) parts, both variables, the voltage and current, have to be matched. The simplest way to do it is to replace each cut branch by dependent voltage and current sources as shown in Figure 17.

The circuits 17a and 17b are equivalent when $i_b = i_a$ and $v_b = -v_a$. $i_a$ is evaluated from Circuit I and used as the current source in Circuit II. $v_b$ is evaluated from Circuit II and used as the negative voltage source in Circuit I.

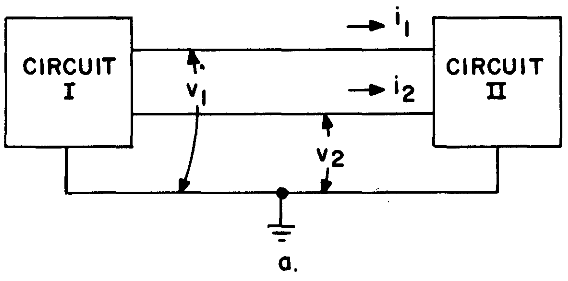


Figure 16a—A circuit

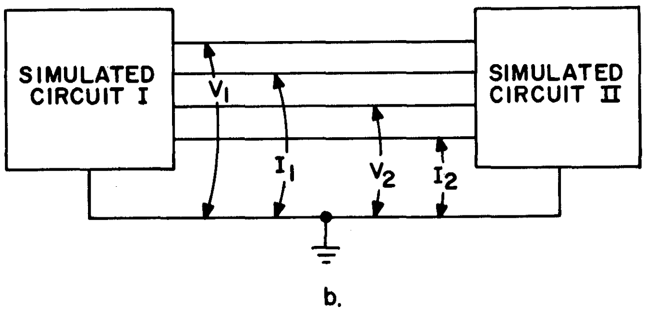


Figure 16b—Simulation of a circuit

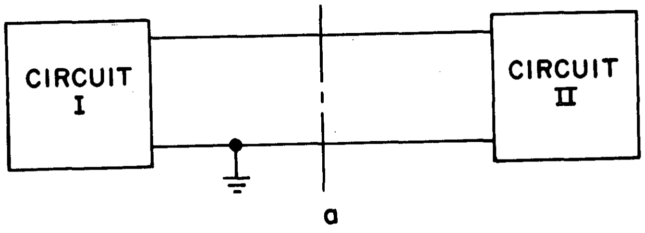


Figure 17a—Circuit

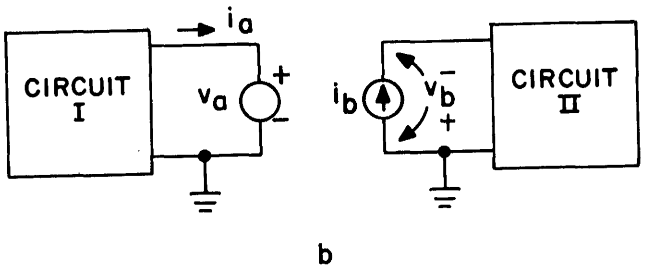


Figure 17b—Partitioned circuit

As an example, let us consider the circuit in Figure 18. The circuit has to be partitioned along the dotted line and simulated as two separate interconnected circuits. The interconnection is replaced by a voltage source $v_a$ and a current source $i_b$ as in Figure 18b. In order for the circuits to be equivalent, $v_b = -v_a$ and $i_a = i_b$.

The simulation of Circuit I and Circuit II and the interconnection between them is shown in Figure 18c.

The choice of using voltage or current sources as terminations of partitioned circuits is sometimes dictated by the topology of the circuits, e.g., a capacitor instead of the inductor in Circuit I will force one to choose a current source as a termination otherwise one state variable will be eliminated.

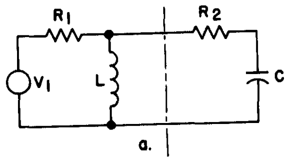


Figure 18a—Circuit to be simulated

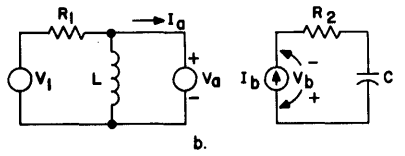


Figure 18b—Partitioned circuit

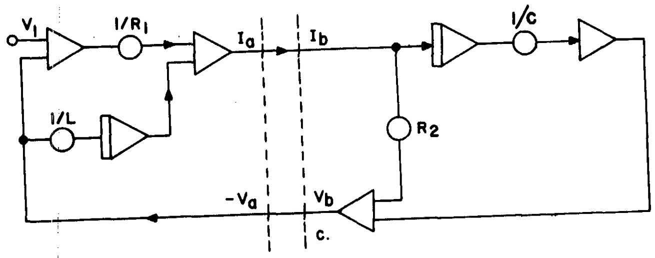


Figure 18c—Simulated circuit

## REFERENCES

1 R D BRENNAN H SANO
*PACTOLUS—A digital analog simulator program*
Proc FJCC Vol 26 1964

2 G A KORN T M KORN
*Electronic analog and hybrid computers*
McGraw-Hill 1964 Chapt 3

3 P HERICI
*Error propagation for difference methods*
John Wiley 1963

4 G A BEKEY W J KARPLUS
*Hybrid computation*
John Wiley 1968

5 L A O' NEILL
*Adaptive detection and representation*

Johns Hopkins Univ 1966 PhD Thesis

6 P BALABAN J LOGAN
*Analog computer simulation of semiconductor circuits*
Proc SJCC Vol 32 1968

7 I W SANDBERG H SHICHMAN
*Numerical integration of systems of stiff nonlinear differential equations*
BSTJ Vol 47 No 4 April 1968

# A time shared I/O processor for real-time hybrid computation

*by* T. R. STROLLO, R. S. TOMLINSON and E. R. FIALA

*Bolt Beranek and Newman Inc.*
Cambridge, Massachusetts

## INTRODUCTION

There are economic advantages to time-sharing a facility with hybrid resources. It is quite unlikely that any single hybrid problem will be able to utilize all of the system resources 100 percent of the time. This is the same kind of reasoning that leads one to consider time-sharing for conventional digital problems. However, time synchronous real-time hybrid time-sharing and non-synchronous non-real-time digital time-sharing are quite different problems, with the former posing some considerable difficulty to sequential digital machines.

A sequential machine can perform only one operation at any single instant of time. Time-sharing is thus accomplished by dividing the available time amongst several tasks, and by task switching from one task to another at judicious times. Most sequential machines cannot perform this switch rapidly. Conventional solutions to real-time problems utilize the central processor of the system for both hybrid computation and I/O; the central processor is usually a sequential machine. These solutions require rapid and frequent switches of the attention of the CPU and, as a consequence, are expensive of machine time and are inaccurate in their timing.

To avoid these disadvantages we have separated the hybrid I/O from the hybrid computation. The hybrid I/O is handled by a special processor called the Hybrid I/O Processor[1] while the hybrid computation is performed by the central processor. This separation gives us a system which can service several simultaneoy real-time hybrid problems efficiently and accuratelt as well as several non-real-time problems concurrent with the hybrid problems.

### *Hybrid problems*

When we speak of real-time demands, we generally visualize applications where the computer system is required to interact with an experiment which is being conducted in the real world in real-time. While these experiments are certainly examples of real-time demands, we extend the definition to include any external demands of the computer system where the reaction time to a demand must be relatively small or where the timing of events must be precise.

## Data generation and acquisition

The simplest demand which fits the real-time category is the data acquisition or data generation problem. Here the computer system is called upon to sample or generate repetitively an analog signal. Data rates as high as 20 KHz are typical for speech sampling or generation. Much lower rates are typical in manual control experiments. Sampling or generation is generally performed periodically at a fixed rate; thus, most of these problems may be considered synchronous. However, in some cases the initiation of the sampling is not synchronized to the computer system. For example, the sampling of a recording on an analog magnetic tape may have to start when a control signal is read off the tape. The computer system generally has no control over this start time, and this sampling problem is now *asynchronous* to the computer system because its initiation cannot be predicted or prescheduled.

### Hybrid simulations

Hybrid simulation problems are also in the real-time category. Generally an analog computer will perform the linear portions of the simulation while the digital computer will perform the non-linear portions. The digital portion of the problem is often characterized by the following synchronous behavior:

1. Sampling a set of input values from the analog computer at a specific time.
2. Performing some digital computation utilizing these input values.
3. Setting up some output values for the analog computer at a specific time.

### Interactive data generation and/or acquisition

Display generation problems are often in the real-time category. This is especially true when displays are used to simulate real world activity such as the view of a runway while landing an aircraft or the view of the instruments of a control panel. Display generation problems are much like any waveform generation problem, but the data rate is generally high, and display problems often tend to present asynchronous demands such as the generation of a new display in response to the changes in a manual control by a human operator.

These three categories of real-time problems classify most common real-time demands.

## Conventional hybird systhms

*Separation of the computation and the I/O*

United Aircraft Facility

Time-shared hybrid systems have existed since 1963 when Belluardo *et al* at United Aircraft Corporation developed a hybrid computation facility[2] based on the DEC PDP-6 computer system. This facility handles several hybrid problems simultaneously, but the problems must have very similar demands. They must be synchronous problems with the same base repetition frequency. The maximum base repetition rate is about 6Hz which is quite slow.

Hybrid I/O is performed by the PDP-6 arithmetic processor in response to I/O commands issued directly by the user programs. These I/O commands are protected in that they cannot affect other users' hybrid devices.

Each of the problems being serviced by the system is assigned a particular start time, offset from the base period start time by a fixed number of real-time clock ticks, which are 1/60 seconds apart. It is assigned as many consecutive "clock tick" slots as needed to ıun. An example of this type of scheduling is shown in Figure 1.

### MIT-ESL-hybrid facility

Connelly developed a hybrid facility[3,4] at the MIT Electronic Systems Laboratory on the DEC PDP-1 computer. This facility also time-shares several hybrid problems which are synchronous in their timing demands. However, the repetition frequencies of these problems are considerably more flexible. The scheduling procedure utilizes a time-slot algorithm with the time-slot duration about 10 times larger than the computer's switching time or about five milliseconds.

The repetition frequencies of the problems must all be members of a "synchronizable set" which means each element of the set is a common multiple of all smaller elements. Such a set of periodic processes is easily scheduled by dividing time up into the highest common denominator of all the process periods. Like the United Aircraft scheduler, a process gets its first time slot at exactly synchronized times, but it may get several temporally disjoint slots before it completes its computation for each basic repetition. An example of time slot scheduling is shown in Figure 2.

The hybrid I/O is performed by the PDP-1 central processor, again in response to protected, direct I/O commands from the user. It is important to realize that with this type of scheduling, the I/O commands must be issued at the start of each repetition period since this is the only time-slot for which the user can

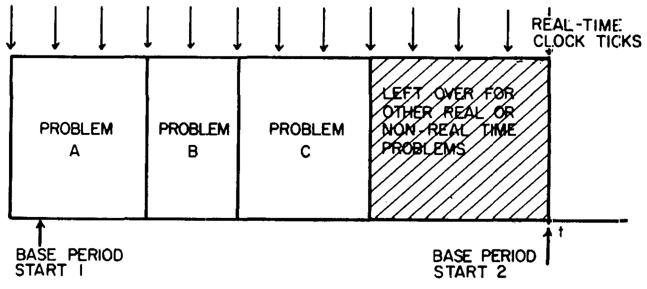


Figure 1—United Aircraft scheduling

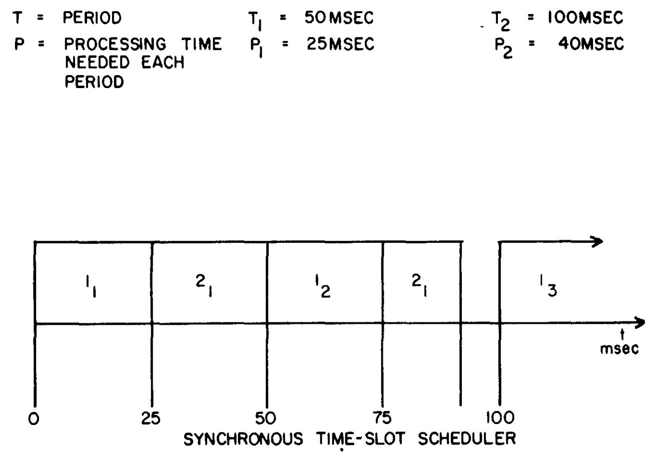


Figure 2—Example of MIT-ESL time-slot scheduling

be guaranteed a specific start time (at the beginning of his repetition period). The relative placement of the other time slots is dependent upon the other problems being serviced by the system.

The conventional hybrid systems to date are represented by the examples given above. These systems will not handle either asynchronous or high rate real-time problems.

## Why conventional systems cannot handle high rate problems

A general purpose digital computer when performing real-time I/O consumes a great deal of its capacity since it was not designed for such a task. A general purpose processor is not well-suited to performing real-time I/O because it cannot switch its attention between tasks rapidly enough. Switching times on the order of a half-millisecond for user processes, and 20 microseconds for servicing interrupt requests are typical of conventional time-shared general purpose processors. This means the overhead of performing real-time I/O with these processors is prohibitive, especially if the real-time I/O rate is high.

A general purpose processor is also not well-suited to accurate timing control of relatively high rate of I/O. Since current central processors are generally asynchronous machines, it is difficult to time operations precisely with a central processor. Further timing skew is introduced by the tendency of general purpose time-sharing systems to get themselves into non-interruptible states for short periods. These occur when the interrupt system is disabled, or higher priority interrupts are processed or the user process's run period is interrupted.

It would seem that processing all real-time I/O on the highest priority interrupt channel triggered by real–time clocks would be the most desirable method for CPU-performed real-time I/O since this seems to minimize skew and overhead. However, one finds it is impossible to make every device the "highest priority" in the interrupt system, and, in fact, there may be other system constraints which force some real-time I/O to be on a lower priority interrupt channel.

Another limitation of interrupt driven real-time I/O is that, in general, one interrupt request will initiate several real-time I/O data transfers. This means the time spent in the interrupt service routine for performing the real-time I/O varies with the number of data transfers to be performed.

These interrupt requests will tend to queue up unless their interrupt service routines are *scheduled* to be non-overlapping.

The precise scheduling of both interrupt–driven and user-initiated real-time I/O is very difficult unless all of the real-time I/O demands are exactly periodic and synchronized to the clock system of the CPU. The source of this difficulty is the inherently slow reaction time of a general purpose processor to random requests for service. Only when all requests are guaranteed to be exactly periodic can they be scheduled in advance, and only then can the CPU be prepared to process them.

We conclude that when real-time I/O is to be performed with precise timing, when the degradation of CPU performance from CPU driven real-time I/O cannot be tolerated, or when both synchronous and asynchronous demands must be handled, real-time I/O should not be performed by the conventional CPU driven methods.

## Separability of real-time I/O and computation

It is clearly possible to design and build a special purpose processor for performing the I/O, but we must first show that it is meaningful and useful to do this. Carefully consider the real-time problem classes previously mentioned. It is evident that most real-time problems place very stringent demands on I/O timing, but less stringent demands on the related CPU computation. That is, the sampling of input data and the conversion of output data to analog signals must be done at precise times, whereas the computation may be done any time after the input operation provided that the output data is ready before it is converted to analog signals. In addition, the I/O operations do not require the full talent of a general purpose CPU, but do require better timing accuracy than the CPU is capable of providing. Therefore, it seems feasible and in fact desirable, to separate the I/O from the computation by providing separate hardware.

### *The BBN hybrid processor*

Our system performs all real-time I/O through a device which we call a Hybrid Processor. It is a special purpose processor with the following important characteristics:

1. Real-time I/O is performed directly between core memory and hybrid devices.
2. The Hybrid Processor is multiplexed among several (up to four) "processes" thus allowing several independent, concurrent hybrid interactions.
3. The switching time between processes is very small (approximately 100 nanoseconds), because the information about the state of a process is small and can be changed rapidly, and because the scheduling of which process to run is performed by the hardware.
4. The time required to perform a single real-time I/O transfer is kept very short (approximately 20 μsec) because of the special purpose nature of the processor.
5. All real-time I/O interactions are handled in a uniform manner, thus requiring no hardware or software changes to incorporate a new hybrid I/O device.

## Command and data structure

The Hybrid Processor implemented for an SDS-940 time-sharing system operates on tables of commands and data. Each command, as shown in Figure 3, is paired with a data word. The nature of the command tells the Hybrid Processor whether the data word is for input or output.

The device type field specifies the types of devices to which the Hybrid Processor talks. These include Analog to Digital Converters, Digital to Analog Converters, Digital level Inputs, and Digital level Outputs. The device number field selects a particular device within a type, such as a channel of the Analog to Digital Converter.

After the command is executed and the data word accessed, the command and data table pointers are both indexed by one, and the remaining word counts for the tables are both decremented by one. The command and data table structure and pointers are shown in Figure 4.

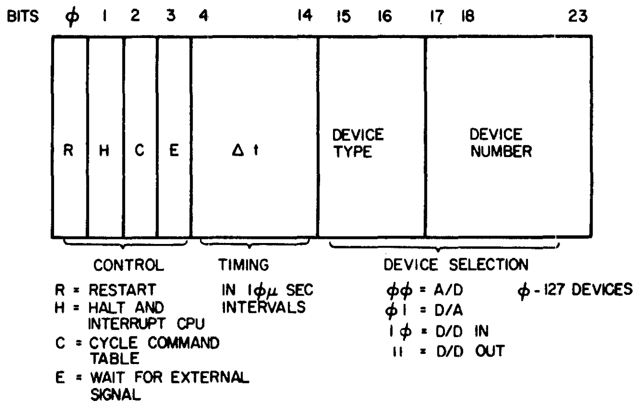


Figure 3—Hybrid processor commands

There is no fixed relationship between lengths or positions of commands and data tables. In fact, most often the command table is quite short and is recycled many times repetitively in order to fill up a large data table. In order to make this cycling operation efficient, some of the control bits are used to initiate the exchange of the cycle pointers with the current pointers. The C bit is used to specify cycling the command table pointers while the condition current data word count = 0 is used to specify cycling the data table pointers.

The Δt field is used to specify the time, in 10 μsce intervals, between the current command and the next command. This value may be thought of as being placed into a clock which is a down-counter operating at 100 KHz. When the counter reaches 0, a request for service is initiated. It is important to remember that

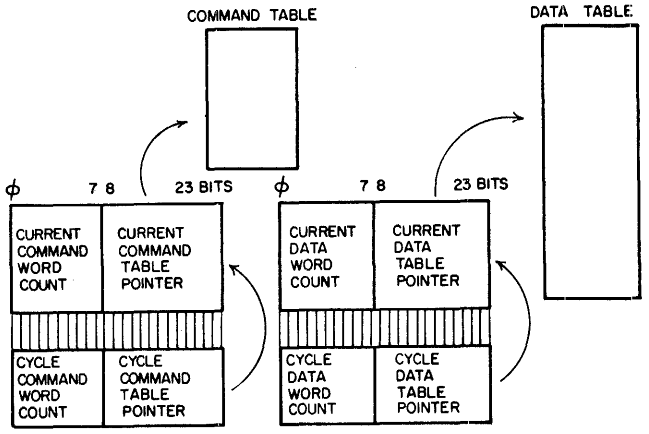


Figure 4—Command and data table structure

the Hybrid Processor is in general running several hybrid processes. When a process requests service, it is quite likely that another request may be in progress or that a higher priority request may be granted first. In order to prevent conflicts from introducing cumulative timing skew, the individual process clocks are designed to count *through* 0 to negative values, and the $\Delta t$ field is *actually added* to the contents of the process clock when the command word is fetched. As long as the result of this addition results in a positive quantity, the process will not be subject to cumulative skew and will be accurately timed on the average. We do a bit better than this, however, by taking advantage of the fact that the clock tick frequency is very accurately crystal–controlled and that the clock = 0 pulse is a very precisely timed event at exactly the $\Delta t$ intervals. This pulse is amplified and available to users for patching. It can be used to initiate a sample and hold gate, for example, or to cause the transfer between buffers in a double buffered D/A converter. This means extremely precise timing control with a *resolution* of 10 $\mu$seconds, and crystal accuracy is achieved.

The remainder of the control bits in a command word are used to stop (H) or restart (R) a process. An external signal control bit (E) enables a temporary stop to wait for a signal external to the computer and the Hybrid Processor to restart the process.

## Connection of hybrid processor to SDS-940

The Hybrid Processor is attached to the SDS-940 via a Data Multiplexor Channel (DMC)[5] with a modified Data Sub-Channel II (DSC II)[1], as shown in Figure 5. The status of a block transfer for a DMC sub-channel is normally contained in one of two "internal interlace words" which are located in fixed adjacent positions in core memory. These interlace words contain the remaining word count in the leftmost bits and current location of the block transfer in the rightmost bits. The economy of using core memory words instead of flip-flop registers is quite important, and this economy is retained by the Hybrid Processor. However, the DSC II has been modified so that the locations of these interlace words are no longer fixed but are uniquely determined by the particular process selected for service. That is, the first process (Process A), uses words n,* n+1 as interlace words; the second process (Process B), uses words n+4, n+5, etc. Also during the command fetch, the even interlace word is

* where n is a memory address which is Ø MOD 4

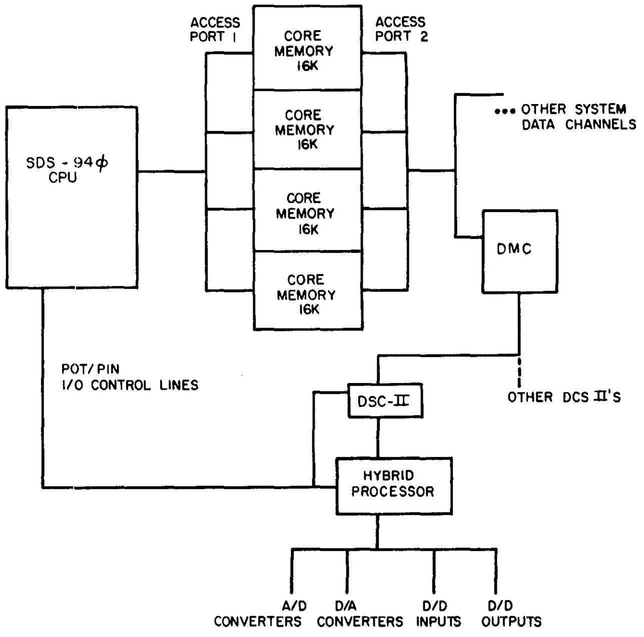


Figure 5—Connection of hybrid processor to SDS-940

used as the current command pointer word, while during that data fetch or store, the odd interlace word is used as the current data pointer word. Note, that as before, the state of the block transfers is completely contained in the memory interlace words.

Each hybrid process can switch command and data tables by using the "cycle" interlace words. Every process actually has four interlace words. Words n and n+1 are the *current* command and data table interlace words respectively. Words n+2 and n+3 are the *cycle* command and data table interlace words. A process can cause the contents of its cycle interlace words to be moved into the current interlace words. This effectively switches the command and/or data tables to new core areas. This switch is accomplished without CPU intervention, but the CPU must establish new cycle words soon after cycling occurs if the next cycle operation is expected to switch to yet another core area. In fact, a safety interlock will abort a process and signal an error interrupt if a cycle attempt is made before the previous cycle operation was properly responded to by the CPU. More details of the BBN Hybrid Processor implementation on the SDS–940 are available in another document.[1]

Suppose the maximum time to service a hybrid I/O request were $\alpha$. This time would be measured from the start of processing of a request by the Hybrid Processor to the completion of this request. Then a conservative estimate of the bandwidth of the Hybrid Processor would be $1/\alpha$.

A simple scheduling technique would involve selling fractions of this bandwidth to users. Each user would buy $1/\alpha_i$ of the bandwidth of the Hybrid Processor, such that

$$\sum_{\text{all } i} 1/\alpha_i \leq 1/\alpha$$

It would now be necessary for the Hybrid Processor (or whatever controls the Hybrid Processor) to insist that user $i$ never perform hybrid I/O faster than $\alpha_i$ seconds between hybrid interactions. This simple check could be performed by either hardware or software. This technique would guarantee that the Hybrid Processor would never be over-committed.

On the SDS-940 implementation of the Hybrid Processor, the maximum $\alpha$ is approximately 20 $\mu$sec.* This means the Processor has a guaranteed bandwidth of 50 KHz.

Simultaneous requests for service are resolved by a simple priority network which selects the highest priority process currently requesting service to run (Process A is highest priority, B, C, D are in order of decreasing priority). This means the higher priority processes see the least instantaneous skew, but none will see any cumulative skew if the bandwidth scheduling rules are adhered to, and none will see instantaneous skew if the clock = Ø pulse and sample/hold gates are used correctly.

## Device and timing protection

The Hybrid Processor commands reference all hybrid devices in an unrestricted manner. If the user is given direct access to these commands, he could detrimentally affect another user's experiment by changing a value on another user's D/A converter, for example. Also, one user could easily lock out Hybrid Processor service from another user if he had a higher priority process and usurped all of the Hybrid Processor's capacity. It is therefore not feasible for the user to construct his own command tables. Instead, the time-sharing monitor constructs these tables for the user and keeps them in monitor core. The monitor makes certain that a user does not access another user's devices or usurp all of the Hybrid Processor's time. The data tables are, however, kept in the user's own address space.

* for D/A conversion, approximately 30 $\mu$sec for A/D conversions.

## Hybrid processor software

Some very elaborate software (with about 2K words of machine language code) exists in our time sharing monitor on the SDS-940 for controlling the Hybrid Processor. This software locks and unlocks pages of data tables into core; sets up the transfers of data table pages from core to drum and vice versa; worries about anticipating pages before they're needed and getting the drum requests on a high priority drum queue; and provides a convenient handle on the hybrid processor for users.

The user interface to the hybrid processor is provided by some SYSPOP's,[5] which permit the user to assign and deassign hybrid devices; assign and deassign hybrid processes; define a sequence of command and data tables to be executed a specified number of times; specify a prototype of the command table which gets set up in the monitor's address space; specify the boundaries of data tables in the user's address space; start and stop processes; and interrogate the status of assigned processes.

## Real-time CPU usage

With a Hybrid Processor I/O system, user programs or user I/O need not be periodic. The I/O can be precisely timed using the Hybrid Processor independent of CPU activity. Therefore, it no longer is necessary to start CPU computation at exact times.

Suppose that for each process the following parameters were specified:[6,7]

1. $T$ The period of the process (exact period if synchronous, *minimum* period if asynchronous).
2. $P$ The *maximum* amount of CPU time the process may require each period.
3. $D$ The *maximum* tolerable delay between the moment the process requests service and the time when all servicing has been *completed* (most synchronous processes would allow service to be completed any time during the period i.e., D = T).
4. Whether the process is synchronous or asynchronous.

Using this characterization, the demand of the process upon the system might be phrased as follows: "When my process requests service it must be granted $P$ seconds of CPU time within $D$ seconds of when the request is made. My process will never request service more often than $T$ seconds after the previous request." The parameters $P$, $D$, $T$, and the specification of whether or not the process is synchronous enables the system to decide whether the demands of this process (and all others) can be successfully met. The system cannot, of course, guarantee service to a set of real-time processes with arbitrary $P$'s, $D$'s, and $T$'s. In fact, two restrictions are obvious:

$$\emptyset < P_i < D_i \leq T_i \tag{1}$$

and

$$\sum_{\substack{\text{all i}\\ \text{processes}}} \frac{P_i}{T_i} \leq 1 \tag{2}$$

If the sum in (2) were greater than unity, it would be possible for the real-time processes to require more than 100 percent of the available CPU time.

The scheduling algorithm used to select which process runs at any time is intimately related to the guarantees which the system can make to a set of users. It would be desirable to find a scheduling algorithm which would allow:

$$\sum_{\substack{\text{all i}\\ \text{processes}}} \frac{P_i}{T_i} \text{ to be close to 1}$$

and would minimize the amount of switching between processes to reduce overhead. It can be shown that if switching time is negligible, no algorithm can do a better job of scheduling for synchronous or asynchronous processes than the following:

*Run the process which must be completed soonest*

That is, whenever a process requests service, the system computes the time when the process must complete service ($\tau_i$), which is equal to the current time plus $D_i$. The system then decides to run the process with the minimum $\tau_i$. Whenever servicing is completed or aborted (for trying to use more than $P_i$ CPU time) the system runs next the process with minimum $\tau_i$. This algorithm and the necessary and sufficient conditions under which the system can undertake to run a set of processes are discussed in detail by Fiala.[6]

## Costs

The Hybrid Processor is not an inexpensive device. Approximately \$20K of digital hardware components are necessary for a Hybrid Processor, not including any of the analog or hybrid equipment. The labor involved in designing and implementing the hardware and software is approximately 1½ man-years. We believe this cost is justified by the utility of the processor.

*Future work*

## Future hybrid processor revisions

Several changes will be made in our new hybrid I/O system for our next research computer (a DEC PDP-10).[8] These changes will increase the total available bandwidth, improve the command/data flow control so that even less CPU capacity will be required to direct the Hybrid Processor, make several improvements to the clock system, etc.

### New clock system

A 36-bit time of day clock will be implemented which counts at 100 KHz. It will be possible to read this time via an I/O input command over the PDP-10 I/O buss. This 36-bit count will recycle approximately every eight days.

At least two 36-bit "alarm" registers will be used in conjunction with the clock. These registers will be compared with the values in the clock after each "tick" settles down. If a match on any register is found, the following events will occur:

A. A CPU interrupt request will be generated so that a new 36-bit value may be placed in the alarm register and any CPU action which was to be initiated at this time will be triggered (such as the scheduling of a new process to run).

B. Each alarm register will have eight enable bits whose set output will be gated with the alarm pulse and this gated result will be buffered and available for patching to trigger external devices.

The control of the alarm registers will require the use of a PDP-10 I/O output instruction to the selected alarm register to set any combination of the eight enable bits followed by a PDP-10 I/O output instruction to the selected alarm register to set up the 36 bits of the register itself.

### Hybrid I/O

The hybrid I/O capability will be quite similar to the capability of our current SDS-940 Hybrid Processor. The channel will operate on command and data tables with each command paired with a corresponding word in the data table.

The command format will also be similar to the current Hybrid Processor on our SDS-940. However, the flow through these command and data tables will be directed by two new tables per process called the command and data flow tables. These will replace the "cycling" operations by "driving" the Hybrid Processor through command and data tables a specified numbers of times. The "cycling" operation had the disadvantage of putting a large burden on the CPU for processes which cycle often (which proved to be true for many processes).

We will also implement the command and data table pointer words in hardware to increase Hybrid Processor bandwidth.

## CONCLUSIONS

The use of a Hybrid Processor permits many real-time experiments which were not possible in the past, and are not possible on other real-time computer systems. We are able to handle high speed as well as asynchronous hybrid interactions. Most of this is made possible by the separation of the real-time I/O functions from the computation function. The real-time I/O functions are performed by a processor especially designed to handle real-time I/O, and the computations are performed by a general purpose processor.

## ACKNOWLEDGMENT

This work was supported by the Advanced Research Projects Agency of the Department of Defense (F19628-68-C-0125).

## REFERENCES

1 T R STROLLO R S TOMLINSON E R FIALA I J ELKIND
*The hybrid processor*
AFCRL-67-0485 BBN Rpt No 1686

2 R BELLUARDO R E GOCHT G A PAQUETTE
*The hybrid computation facility at United Aircraft Corporation Laboratories*
Proc DECUS 1963 Maynard Mass 261-269 1964

3 M CONNELLY
*Preliminary design of a time-shared, real-time, simulation facility*
Memo No 1 MIT ESL-DSR 76259 Dec 19 1966

4 M CONNELLY
*Preliminary design of a time-shared, real-time, simulation facility*
Memo No 2 MIT ESL-DSR 76259 Jan 30 1968

5 SDS-940 Reference Manual 900640A
Scientific Data Systems Aug 1966

6 E R FIALA
*Scheduling of real-time processes in a time-shared environment*
MIT Masters Thesis 1968

7 M S FINEBERG O SERLIN
*Multiprogramming for hybrid computation*
Proc FJCC 1967

8 DEC PDP-10 System Reference Manual
HGAA-D June 1968

# On-line software checkout facility for special purpose computers*

*by* J. S. HUGHES

*IBM Corporation*
Huntsville, Alabama

and

T. H. WITZEL

*IBM Corporation*
Gaithersburg, Maryland

## INTRODUCTION

An on-line software checkout facility for special purpose computers (referred to as the Flight Software Development Laboratory) has been created to aid programmer/engineers in the development of programs that will operate in a spaceborne computer aboard the Apollo/Saturn IB and V Launch Vehicles. The Flight Computer operates as an integral part of various vehicle subsystems in the Instrument Unit (IU). The subsystems provide onboard navigation, guidance, control, sequencing, data compression, and ground communications. These functions are illustrated in Figure 1. Continued emphasis is placed on error-free flight software, since it is an essential element in overall vehicle performance. No opportunity exists to test or exercise the flight program in its actual flight environment prior to a mission. Therefore, to ensure the integrity of the flight program, simulators are used to accomplish flight testing. The purpose of this paper is to present the organization of one such simulator that has been created for the sole purpose of the development and checkout of Saturn flight software. The emphasis throughout the design and implementation of the Laboratory has been that it must be user-oriented for program checkout. Before the existence of the Laboratory, available facilities for checking out flight programs were oriented to hardware checkout. Although such facilities can be, and have been, rigged for program checkout, they have not provided the type of assistance required to produce the quality of software demanded by spaceborne computers. The Laboratory is believed to be unique in the capabilities it provides to the programmer/engineer in controlling and affecting the operation of the Flight Computer in a real-time environment.

Flight software development begins with a set of explicit engineering requirements: equation and logic definition, range of variables, and expected performance data. After an intensive analysis of the requirements, the flight software is designed and organized to meet these engineering requirements with minimal flight computer memory and reasonable flexibility. After the flight program has been flowed, scaled (fixed point computer), coded, assembled, and checked out by the program unit or module, the flight phases are integrated and checked out. This process continues until the entire flight software has been integrated. The procedure described above requires that the programmer/engineer be able to measure and evaluate his progress in an efficient manner. The purpose of this laboratory facility is to provide the programmer/engineer with a user-

* This work was performed under contract with NASA's Marshall Space Flight Center.

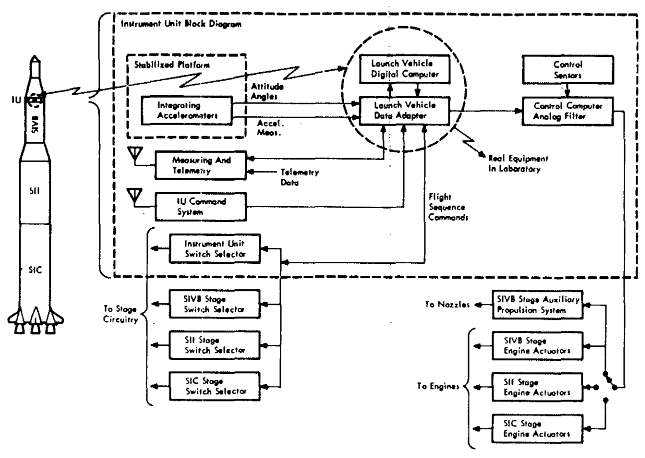

Instrument Unit Block Diagram
IU
SIVB
SII
SIC
Stabilized Platform
Integrating Accelerometers
Attitude Angles
Accel. Meas.
Launch Vehicle Digital Computer
Launch Vehicle Data Adapter
Control Sensors
Control Computer Analog Filter
Real Equipment In Laboratory
Measuring And Telemetry
Telemetry Data
IU Command System
Flight Sequence Commands
Instrument Unit Switch Selector
To Stage Circuitry
SIVB Stage Switch Selector
SII Stage Switch Selector
SIC Stage Switch Selector
To Nozzles
SIVB Stage Auxiliary Propulsion System
To Engines
SIVB Stage Engine Actuators
SII Stage Engine Actuators
SIC Stage Engine Actuators

Figure 1—Real and simulated flight equipment

oriented tool by which he is able to test and evaluate his programs in a simulated flight environment, using an actual spaceborne computer and interface hardware. This enables him to measure and evaluate flight software performance against the engineering requirements for the many vehicles and environmental variations.

The Laboratory user must produce quality software in the shortest possible time frame. The key objective in designing the Laboratory was to provide accurate simulation models in the form of user-oriented tools. Thus, the Laboratory user can swiftly determine the progress and results of his work through real-time man-computer interaction. The computer offers data, counsel, and guidance to the man, who in return supplies certain indispensable knowledge of the overall system. Systems reliability and effective communications between the Laboratory and user play a major role in establishing user confidence. Operating experience in the Laboratory has clearly demonstrated that these objectives have been satisfied.

### *Hardware configuration*

The Laboratory has as its main hardware components an IBM System/360 Model 44, linked through a special purpose interface to a Saturn Launch Vehicle Digital Computer and Launch Vehicle Data Adapter. An IBM 2250 Display Unit is employed as an integral part of the Laboratory, providing two-way man-computer communications. Figure 2 illustrates the organization of the hardware components and in general indicates the basic paths of information flow.

One high speed multiplexer channel has been dedicated to the flight hardware interface. Each of the subchannels is likewise dedicated, as shown in Figure 2. The dedicated channel and subchannels minimize

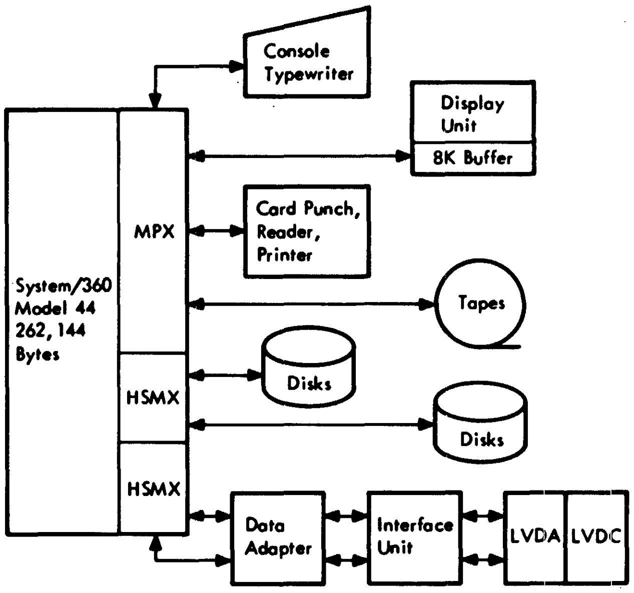


Figure 2—Flight software development laboratory—Block diagram

interference from other I/O activities and enable the creation of a special low overhead channel scheduler. These features incorporated with the 32-level priority interrupt scheme make the Model 44 highly responsive to the real-time interface requirements. The other high speed multiplexer channel is dedicated to disks that support real-time data collection and permit fast access for the display system.

In this particular application, six of the 32 levels of priority interrupt are used by external hardwired equipment. The others are used by internally generated software functions for scheduling time-dependent software functions.

The Launch Vehicle Digital Computer and Launch Vehicle Data Adapter are the two flight components that have been integrated into the Laboratory.

The Flight Computer is a general purpose computer which, under control of a stored program, processes data serially, using fixed-point 2's complement arithmetic.

The Launch Vehicle Data Adapter serves as an input/output device for the Flight Computer and the central station for the signal flow in the Saturn Astrionics System, which is illustrated in Figure 1. The Data Adapter accepts discrete input signals from the stage switch selectors, Instrument Unit command receiver, ground launch computer, telemetry computer interface unit, telemetry data multiplexer, control distributor, and other vehicle equipment. It has output registers to provide discrete output signals to the above-mentioned equipment. It also accepts and processes computer interrupt signals from the ground launch computer and Instrument Unit equipment.

The interface unit provides all the normal ground and flight communications paths between the flight hardware and the central processor. However, this interface was designed to go beyond these requirements. The interface is unique in that it was designed to place emphasis on (1) minimizing the central processor interface traffic and (2) maximizing user visibility by giving the user the control of internal flight hardware operations and the access to information internal to the Flight Computer. Also, the unit was designed for ease of maintainability. Specifically, three major capabilities have been incorporated into the interface unit. First, the interface unit has been designed so that it can control the internal operation and timing of the Flight Computer and Data Adapter. Secondly, the interface contains special hardware, oriented toward supporting flight program debug as opposed to program verification, which is an independent program audit function performed using the debugged programs. Finally, the interface unit has been designed so that extensive automatic diagnostics can be run from the central processor to isolate suspected interface failures.

The IBM 2250 Display Unit is organized around a cathode ray tube on which computer-programmed graphic and alphameric information is displayed at high speeds. This provides visual communication between the computer and the user. In addition, keyboards and a light pen provide the user with a versatile means of entering and modifying computer information. With the display system, the user has direct and rapid access to stored data which can be selected, processed, modified, and displayed in alphameric and graphic representation. For example, the user can display and modify memory in both the Model 44 and the Flight Computer through the display unit.

The display unit was configured to minimize central processor time and core requirements on the Model 44. A primary feature of the display unit is a buffer storage of 8,192 bytes, which is used to store images for display regeneration purposes. The use of a buffer enables the display unit to operate concurrently with the computer system, freeing the main core and the channel for other functions. Additional features which greatly compress the image storage requirements are the absolute vector and character generator features.

### *Operating system*

The operating system for the Laboratory is designated as the Checkout Control System (CCS). It is the operating system which is furnished with the IBM System/360 Model 44, with additions and modifications to convert the system from a sequential batch job processor to a real-time multiprogramming processor. However, all the original functions and features have been retained. Programs not requiring the elements of a real-time multiprogramming system may operate as though the additional facilities were not present.

The principal area of the Model 44 Programming System (44PS) in which additions and changes have been made is the supervisor. The required functions of CCS include the ability to support various operations of computing at precise intervals of time. These operations are selected by a priority scheme which controls the sequence of execution. Other operations are designed to execute as a result of interrupts induced outside the central processor. These are generally of such importance that their priorities are higher than operations initiated as a result of time. The function of multiprogramming through a scheme of priority interrupts and the requirement of real-time operation are the principal requirements for CCS. To satisfy these requirements, capabilities in three principal areas have been added. These are multiprogram scheduling, real-time input/output scheduling, and application program phasing control.

A principal element of the program scheduling facility for CCS is the timer queue (Figure 3). It consists of a string of items ordered in ascending sequence of time-to-execute. Each item of the queue contains a pointer to the routine to be executed at the corresponding time. When the timer interrupt occurs, the timer processor routine gains control and the routine corresponding to the timer interrupt is placed into a state of execution. Its immediate or deferred execution is a function of priority levels. When a timer interrupt occurs, a comparison is made between the priority level of the routine currently in execution and the level of the routine for which the timer interrupt has occurred. If the level of the current routine is higher than or equal to the other, it resumes execution while the execution of the lower priority routine is deferred. Conversely, if the priority level of the current routine is lower, the other is placed immediately into execution, temporarily suspending the first. This method of scheduling uses the hardware priority interrupt system and additional software of CCS.

Figure 4 illustrates some of the conditions which may occur with a typical combination of timer-initiated priority routines. Notice that the execution priority

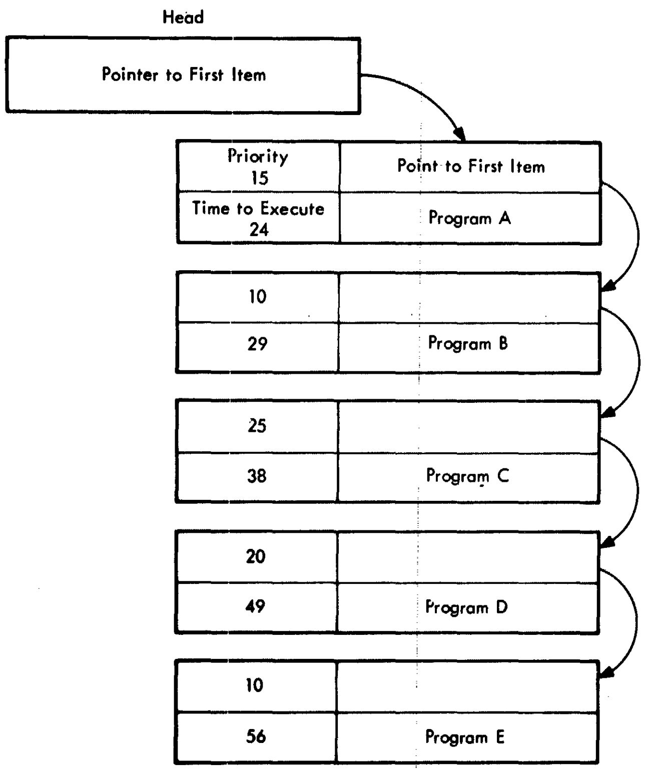


Figure 3—Timer queue

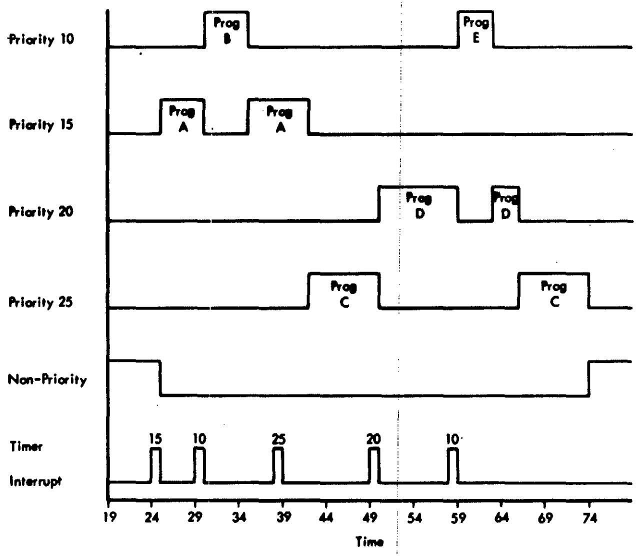


Figure 4—Timer-initiated multiprogramming

level of timer interrupts is coincident with the priority level executing at that time. In addition, the figure shows how the high priority routine gains control from a lower priority routine. In this priority system, low magnitude numbers correspond to high level priority.

Figure 3 illustrates a timer queue containing several items which will initiate programs on various levels at different times. These items match the information illustrated in Figure 4. As each item reaches the top of the list, the internal interval timer is set to the increment of time from "now" until the program is to execute. When the timer expires, a priority level request for the program is set, the item is removed from the queue, and an interval for the next item is calculated. The program pointed to by the item which caused the timer interrupt is attached to its priority level for execution. When the queue becomes empty, the nonpriority level regains control.

The second major feature of program scheduling is the supervision of priority interrupts by the priority interrupt executive. Certain 'housekeeping' functions are performed by this feature, such as register saving and restoring, as control passes up and down the priority levels. Control is automatically given to the priority interrupt executive whenever any one of the 32 levels is activated. The routine to be given control is determined, registers are saved as required, and a pointer to parameters is set. Control is then given to the priority routine. When the routine concludes its operation, it returns control to the priority interrupt executive which restores registers and causes the routine on the next highest level to resume or begin execution.

Figure 5 illustrates the overall flow of data and control in CCS. Whereas Figure 4 illustrates the effect of program scheduling, this figure illustrates the mechanics involved. A program currently executing may be interrupted by the timer (1). The timer processor selects data from the queue (2) and attaches the routine to execute (3). It sets a new interval in the timer (4) and initiates a priority interrupt (5) (assuming the routine is of a higher priority than the current program). The priority interrupt executive determines the routine to execute (6) and gives control to the routine (7) which returns control (8) when finished. The executive then returns control to the interrupted program (9). At step 5, the condition may exist that the timer-initiated routine is of lower priority than the current program. If so, the timer processor returns control directly to the current program (10).

Both application and system programs may queue routines using the timer queue (11). The actual queueing is done by a system routine.

In Figure 5, the dash line connecting "programs" and "current program" is intended to show that the

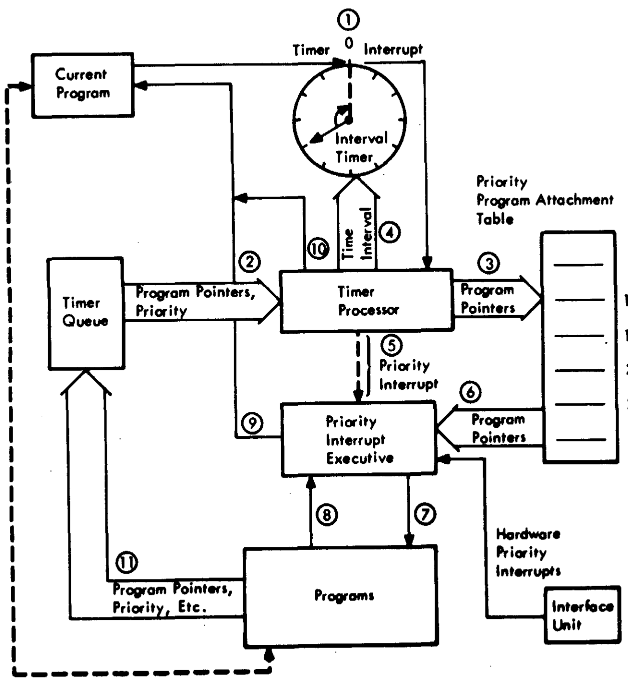


Figure 5—Program scheduling

current program is merely one of many which is selected by the priority interrupt executive.

The 44PS I/O channel scheduler was changed to handle a multiprogrammed environment. Among the changes were new real-time, core resident I/O device routines, a gated front end to the channel scheduler and I/O termination interrupt routine, and de-queueing logic at the exit point of the channel scheduler.

Of several options available to modify the channel scheduler, one was chosen which allows only one transaction into the scheduler at a time. The effect of this method is to allow an I/O request to enter the scheduler and be serviced only if the scheduler is not currently processing a previous request. Due to multilevel program execution, an I/O request being made while another is being processed can occur only when the new request is of a higher priority level routine. Therefore, the request in process when interrupted by a higher priority level is resumed at its point of interrupt. The new higher priority level request is serviced immediately thereafter.

A posting function is associated with I/O termination. Its purpose is to allow priority level routines to request I/O, give up control on their level, and then regain control when the I/O is completed.

To meet the demands of real-time I/O for the Flight Computer, a special low overhead channel scheduler is used. The gated channel scheduler using 44PS has some features neither necessary nor required for the Flight Computer channel.

The CCS phasing function provides the capability to load and initialize the required set of application programs under the control of operations initiated at the display console. An application core load (phase) contains all programs required in memory at the same time to perform one of the major simulation functions.

The phases are resident on the system residence disk volume and are transferred to the application program area of central processor memory when requested by the flight programmer/engineer via the display console. The transfer is implemented by CCS through standard 44PS load capabilities. After a phase has been loaded, the unused portion of memory is calculated and designated as available work space which will later be used by the phase programs. The phase is given program control at its entry point on the non-priority level of program execution.

Selection of the phase to be loaded and executed is made from the initial tutorial display, which is setup by the initial loading of CCS. The user at the display console makes his selection of the phase via the light pen instrument. The selection of a phase initiates CCS operations which result in a core load of the application program area. When the user chooses to change from the execution of one phase to another, he requests the redisplay of the initial tutorial. Upon this request, CCS executes an orderly shutdown of the activities in process for the current phase and then reloads the central processor memory with the phase requested. Figure 6 illustrates the concept of phased program loads and the general allocation of core for the application programs.

### *Application software*

Application software in the Laboratory is designed to perform four basic tasks: (1) hardware diagnostics, (2) flight simulation initialization, (3) flight simulation execution, and (4) post-flight data reduction. A self-contained set of software programs, called a phase, has been constructed to perform each of these tasks. At any given time only one phase resides in core, with both the communication region and temporary data set residing on a disk device.

The hardware diagnostics phase contains programs which perform the power-up and initialization function for the Flight Computer and its interface unit. The diagnostic programs are required for maintaining and servicing the interface unit.

The flight simulation initialization phase consists of the programs to specify the details and options of the

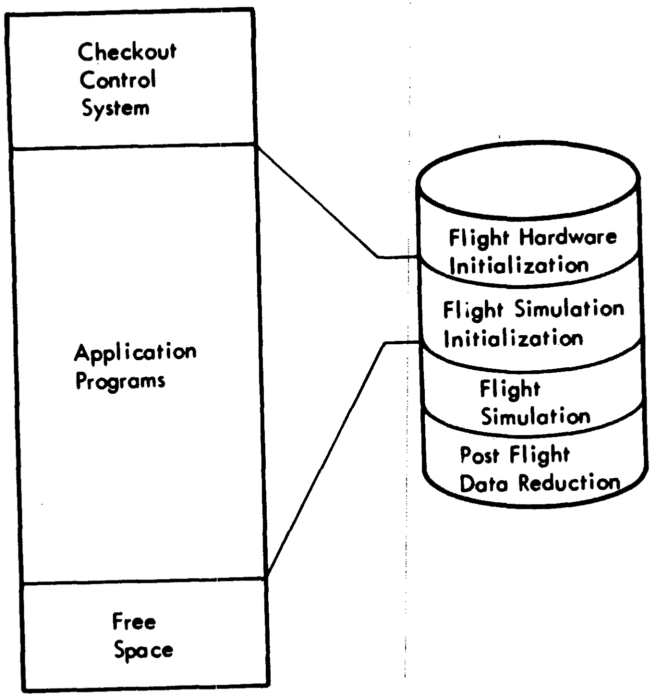

Checkout Control System
Application Programs
Free Space
Flight Hardware Initialization
Flight Simulation Initialization
Flight Simulation
Post Flight Data Reduction

Figure 6—Phase load core memory map

particular flight simulation the user wishes to make. He may specify such items as loading, modifying, and accessing the flight program; digital command system orders; computer interface unit measurements; real-time output quantities; flight pause points; data to be saved for post-flight analysis; the particular Saturn vehicle to be simulated; and the type of simulation run to be made.

The post-flight data reduction phase contains the software necessary to process data that were collected during execution of the real-time simulation phase. This consists of data from the Flight Computer, the 6-DOF simulator, and the FORTRAN flight program model. The capability of generating plots on the display unit is also provided, along with conversion, formatting, analysis, and outputting of data on the printer.

In these three phases, very little use is made of the priority interrupt feature on the Model 44 as an instrument of real-time operation. Practically all programs are initiated by operator action at the display console, and programs receiving control operate on the priority level assigned to display control. This same level is reserved for display control in each phase. However, the entire real-time simulation phase is built around and is controlled by the priority interrupt feature.

The application programs in this phase perform the following tasks:

- 6-degree-of-freedom (6-DOF) launch vehicle simulation.
- Digital command system simulation (ground data link).
- FORTRAN equation and logic model execution of the flight program.
- Data reduction and analysis.

Figure 7 presents an overview of the major application software components required for the real-time phase and their interrelationships. The operator, seated at the display control unit, communicates with the system through preformatted tutorial displays. Three data sets are used as an input interface between the display unit and the real-time application software. The flight and vehicle specifications data set is used to structure the vehicle and flight program skeletons to any of the many missions under development. This data set is defined and generated during the initialization phase. The two remaining data sets (vehicle perturbation and data reduction analysis and control) are accessible both during the initialization and real-time simulation phases. The vehicle perturbation functions allow the operator to specify various vehicle anomalies such as thrust perturbations, command receiver failures, staging or event failures, inertial platform failures, etc., in addition to the start time and duration for each. The appropriate control information is ordered by time of occurrence and recorded in a

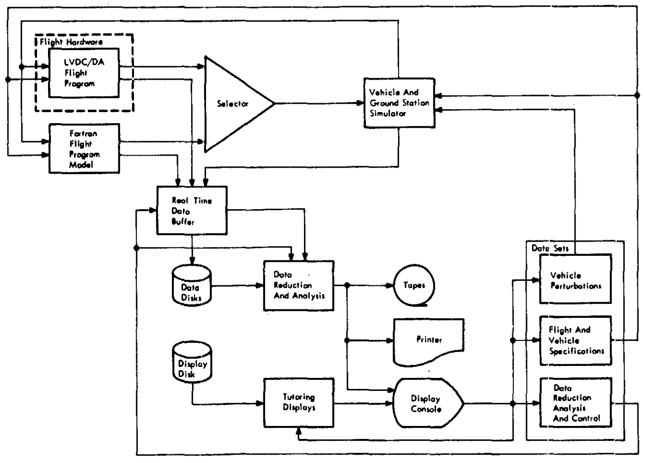


Figure 7—Application software functional flow—Real-time phase

vehicle sequencing queue until the specified activation time. A similar procedure is followed in the creation of the data reduction analysis and control data set.

The FORTRAN flight program model is an engineering representation of the flight program. It serves as an additional reference to measure and evaluate actual flight software performance.

The real-time data buffer receives data from the FORTRAN flight program model and the 6-DOF vehicle simulator as well as telemetry data from the flight hardware. This entire set of data is recorded on tape for the post-flight data reduction phase. Data selected for real-time observation is organized, formatted, and recorded on the disk. This particular data is accessible at any time upon request from the display console as either tabular data or graphic plots. Such displays may be generated from historical data beginning at some particular point in the past and carried up through the current values, or it may start with current values. In both cases, the display is continually updated from the real-time data buffer. In addition, the real-time tabular data may be permanently recorded on the printer.

The 6-DOF simulator consists of both rotational and translational dynamics as well as a simulation of the vehicle subsystems involved in vehicular control, such as sequencing, digital command system, etc. This simulator may be driven by either the actual flight hardware or the FORTRAN flight program model. In turn, the 6-DOF simulator supplies inputs to both the flight hardware and the FORTRAN flight program model.

Each of these real-time application programs is assigned a relative priority and an absolute priority level. Figure 8 shows groups of application programs and their priority interrupt level assignments used in this system at the present time. In general, high priority levels have short execution times. These routines respond to discrete external events or internal keying by the interval timer. Routines with longer execution times are on lower priority levels. Among these routines are real-time reduction and graphic support. The priority level assignments, both hardware and software activated, can be changed easily to optimize system performance.

The requirements of the real-time application programs guided the design of the program scheduler in CCS. As a result, the priority scheduler provides the real-time application programs with a highly flexible operating environment, making the following system attributes possible: First, the System/360 Model 44 and the Flight Computer operate asynchronously with respect to one another. This condition relieves the system of several constraints in its operating environment, which, if present, tend to constrict the system. Second, the application programs are very responsive to the information supplied from the interface unit via the priority interrupt feature. There are six high priority levels of the thirty-two which are assigned to signals from the interface unit. This structure permits immediate response to Flight Computer conditions in the Model 44 by interrupting programs operating on a lower priority level. Third, a related function to point two is that the low priority operations (such as servicing display unit operations) execute on a noninterference basis with the time critical functions on higher priority level assignments. Fourth, with respect to time-slicing, it is a self-adjusting system. This means that programs on lower priority levels will automatically give up time to programs operating on higher levels. For example, the solution rate on vehicle navigation can be changed by simply altering one constant, which will result in the self-adjustment of the system to the new solution rate.

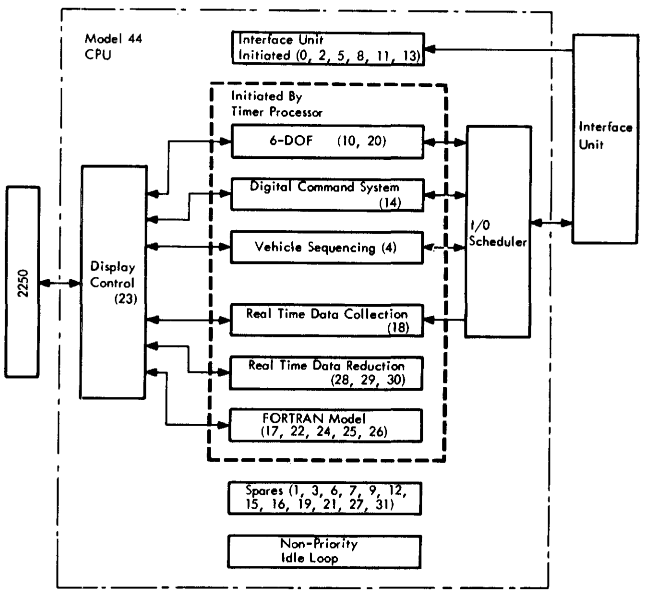

Model 44 CPU
Interface Unit Initiated (0, 2, 5, 8, 11, 13)
Initiated By Timer Processor
6-DOF (10, 20)
Digital Command System (14)
Vehicle Sequencing (4)
Real Time Data Collection (18)
Real Time Data Reduction (28, 29, 30)
FORTRAN Model (17, 22, 24, 25, 26)
Spares (1, 3, 6, 7, 9, 12, 15, 16, 19, 21, 27, 31)
Non-Priority Idle Loop
2250
Display Control (23)
I/O Scheduler
Interface Unit

Figure 8—Functional groupings of priority interrupt line assignments (0-31) in real-time phase

*User/system interaction*

With reference to user/system interaction, the system may be said to have two primary objectives: to provide a more detailed and complete checkout of flight programs, and to ease the burden and reduce the time required of a flight programmer to checkout a flight program. To meet these objectives, the following system criteria were established:

1. Minimal knowledge of the central processor required of system users.
2. Minimum number of people required to run a simulation.
3. Centralized operator control stations.
4. Maximum influence on the flight program by the user.
5. Minimum time required to setup runs.
6. Entire simulations run by nontechnical operators.

In order to satisfy these criteria, system start-up procedures were automated, peripheral device management routines were written (to allow tapes and disks to be remounted on arbitrary drives), direct access storage was fully utilized, program overlay was used extensively, and all operator control (after initial setup) was centralized at the display unit.

The graphic display software provides the interface between the application programs and the IBM 2250 Display Unit through which the user communicates with the system. Through this software, the display console operator is in complete control of the flight program and has a very wide range of capabilities in initializing, controlling, monitoring, and analyzing the flight program performance.

The display program that provides this interface operates in a real-time environment. Therefore, to reduce the core and time requirement necessary to create each individual display, all displays are preformatted by an off-line graphic program. (See Figure 9.) This program receives card images of the text and control information associated with each display and creates a 'book' of displays. This display book resides on a disk cartridge and is divided into one index and as many chapters as there are displays. Each display text is in an 'expanded' format containing embedded graphic orders and in a format ready for transmission to the display unit buffer. It requires no editing, scanning, or unpacking in real time. The keyboard and light pen pages provide control information needed by the real-time display control program to respond to operator keyboard and light pen inputs. For each chapter created by the off-line program, there is a corresponding entry in the index. Each index entry contains the name and disk address of its corresponding chapter.

During system initialization, the real-time display control program reads the index into core and retrieves the system initial display from the display book. The

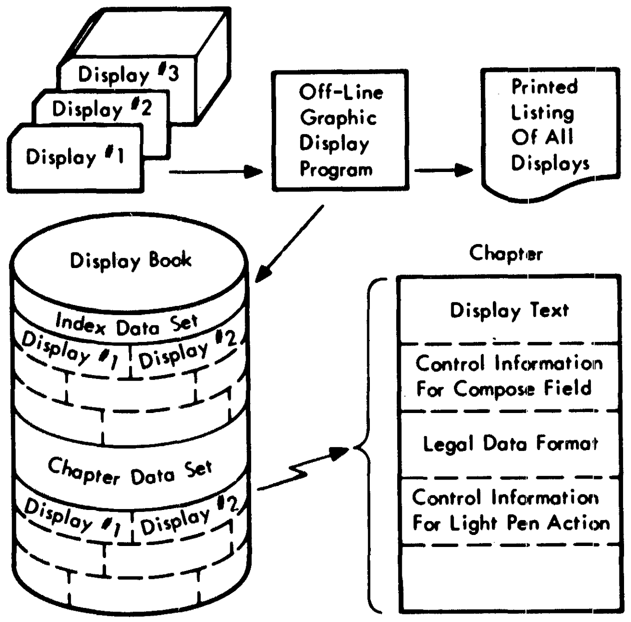


Figure 9—Display book generation and organization

initialize phase of the real-time display software is complete when the initial options are displayed. Light pen or keyboard inputs from the display console operator are required to initiate the display of new texts.

The display device routine services the light pen and keyboard I/O interrupts and schedules the display control program on the timer queue for immediate execution on a predetermined priority interrupt level. When display control gets control on its priority interrupt level, the type of action taken by the operator is examined. The light pen page and the keyboard page of the current display chapter define all legal light pen inputs and keyboard entries.

The display control program displays the proper text in response to the operator's light pen actions, validates keyboard inputs, and passes control information specified in the keyboard or light pen page. Response to the user's inputs appears to be instantaneous to the operator (500 milliseconds maximum). When a longer time is needed to process the operator's request, the program to perform the operation is scheduled to operate on another priority level and normal display processing continues. The operator may initiate several tasks to be performed simultaneously.

On a light pen detect, the light pen page of the current display chapter is examined for a possible NEXT PROGRAM. If one is specified, control is passed to it and the NEXT DISPLAY is presented when the program returns control to display control. When the

NEXT PROGRAM is omitted from the light pen page, the NEXT DISPLAY is presented immediately and the display control program priority level is freed for additional operator inputs.

Display control makes legality checks on all keyboard input against the legal data in the keyboard page of the display chapter and passes the data to the designated program. When it is necessary to input a large amount of data through the display compose fields, or when many displays and light pen actions are required to initiate a procedure, the light pen options and keyboard entries may be predefined on cards or disk. The display software can initiate one option after another and each time return to the predefined option set for another option rather than waiting on the console operator for further action. This speeds the setup for procedures done repetitively and greatly reduces the possibility of operator errors.

The real-time display control program accepts inputs from the display console operator and also from the application programs. While there can be many application programs providing input during a small interval of time, there can be only one display being presented to the operator. The inputs affecting future or past displays are entered in a queue and may be viewed by the operator by use of the function keyboard.

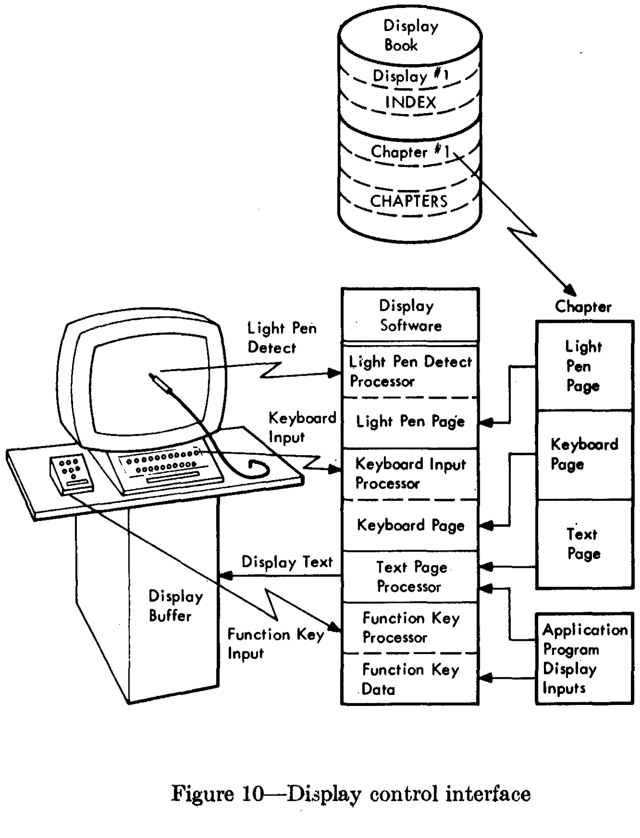


Figure 10—Display control interface

An application program on any priority interrupt level may use the display system to communicate with the user through previously defined input areas in the display text. These input areas may be defined by the user to suit his needs and to present his input data in an easy–to–read format. For example, if the input areas in a display are defined in a column format, the programmer's data will automatically be presented in a column format when the input areas are filled.

Figure 10 illustrates the display control interface with the display unit. The control information pages of a display chapter will remain in memory as long as the display text is being presented to the operator. When the operator uses the light pen or keyboard, display control will use these pages to determine the NEXT PROGRAM and NEXT DISPLAY. When the chapter for the next display is retrieved from the display book, the text page processor merges the display text with any application program data to be displayed and transfers the combined text and data to the display buffer. The new light pen and keyboard pages will remain in memory to identify the next operator action.

A function key must be lit by the function key processor before the key becomes active. An application program can direct display control to activate a function key and present a given control display when the operator uses the key.

As the user views the system, the heart is the display system. The programmed book of tutorial displays is provided to give him complete control over the Flight Computer, the interface, and the simulation itself. The book along with the use of the light pen and the display keyboard leads the user through the functions of powering up the Flight Computer and Data Adapter, loading and accessing the Flight Computer memory and registers, setting up and executing the simulation, and post-processing simulation data. Each user option is carefully spelled out, and all user input is verified before it is accepted by the system. Should error conditions occur (due to incorrect input, hardware failure, or flight program failure), error messages are presented to the user with instructions as to the recovery action.

A complete history of user actions at the display unit is logged on the console typewriter for later reference.

Figures 11 and 12 demonstrate user activity at the graphic display terminal. By using the light pen, the user is able to travel through the display structure illustrated in Figure 11. Figure 12 depicts photographs of the displays represented by the structure of Figure 11.

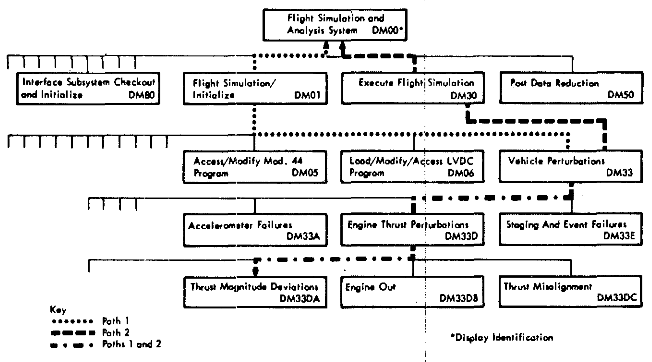
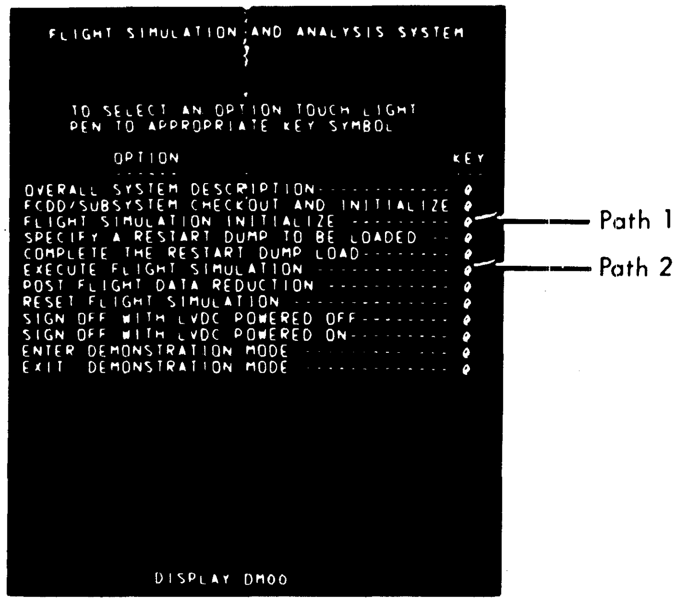

Flight Simulation and Analysis System DM00*
Interface Subsystem Checkout and Initialize DM80
Flight Simulation/ Initialize DM01
Execute Flight Simulation DM30
Post Data Reduction DM50
Access/Modify Mod. 44 Program DM05
Load/Modify/Access LVDC Program DM06
Vehicle Perturbations DM33
Accelerometer Failures DM33A
Engine Thrust Perturbations DM33D
Staging And Event Failures DM33E
Thrust Magnitude Deviations DM33DA
Engine Out DM33DB
Thrust Misalignment DM33DC
Key
Path 1
Path 2
Paths 1 and 2
*Display Identification

Figure 11—Display structure



```
FLIGHT SIMULATION AND ANALYSIS SYSTEM


TO SELECT AN OPTION TOUCH LIGHT
PEN TO APPROPRIATE KEY SYMBOL

       OPTION                              KEY
       ------                              ---
OVERALL SYSTEM DESCRIPTION------------- ●
FCDD/SUBSYSTEM CHECKOUT AND INITIALIZE ●
FLIGHT SIMULATION INITIALIZE ---------- ●
SPECIFY A RESTART DUMP TO BE LOADED --- ●
COMPLETE THE RESTART DUMP LOAD--------- ●
EXECUTE FLIGHT SIMULATION ------------- ●
POST FLIGHT DATA REDUCTION ------------ ●
RESET FLIGHT SIMULATION --------------- ●
SIGN OFF WITH LVDC POWERED OFF--------- ●
SIGN OFF WITH LVDC POWERED ON---------- ●
ENTER DEMONSTRATION MODE -------------- ●
EXIT  DEMONSTRATION MODE -------------- ●



            DISPLAY DM00
```

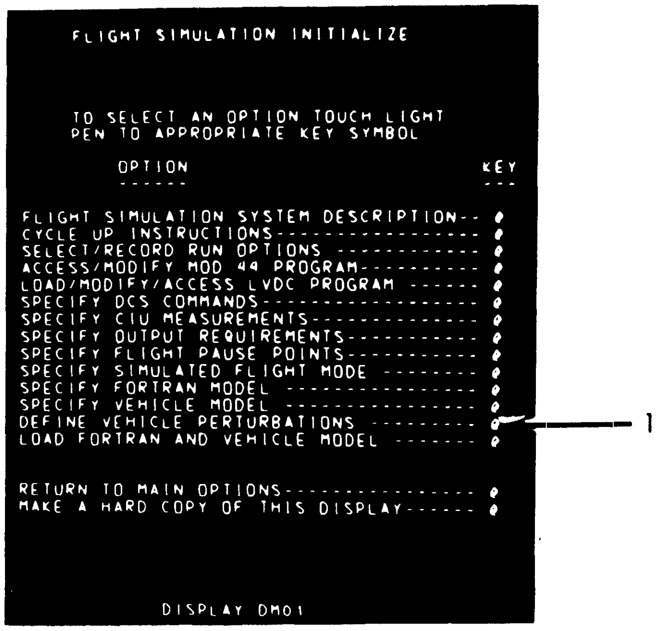
12a

Figure 12 shows the top level display (DM00). When the light pen is applied to the keys for path 1, it leads to a display, DM33DA (Figure 12e), calling for information to be entered from the keyboard. The keyboard entry or compose field is defined by the legend on the display. Path 2 shows how the same path may be entered from the execute flight simulation display, real-time phase.

Along with the capability to setup and execute complete runs through the display system, the user has the ability to monitor the execution and take restart dumps. These, too, are controlled from the display console. A request may be made to printout specific quantities on the printer as they are calculated in either the Flight Computer or the central processor. At the same time he may request various status displays, tabular displays, or data plots to appear on the display unit. Should an irregularity be detected, the operator has the ability to pause the simulation, process all the data acquired thus far, and make changes or corrections. He then has the option to continue the simulation, restart the simulation from various points where restart information is available, or terminate the run entirely.

Because of the complexity of the flight programs, the flexibility of the facilities of the Laboratory, and the desire to ease the burden of job setup as much as possible, a scheme has been implemented to sequence automatically from start to finish. The sequencing and input information can be saved on cards or in data sets on direct access storage. Through the use of this scheme, it is possible for a flight programmer to setup complex runs and submit the job for running by an operator.

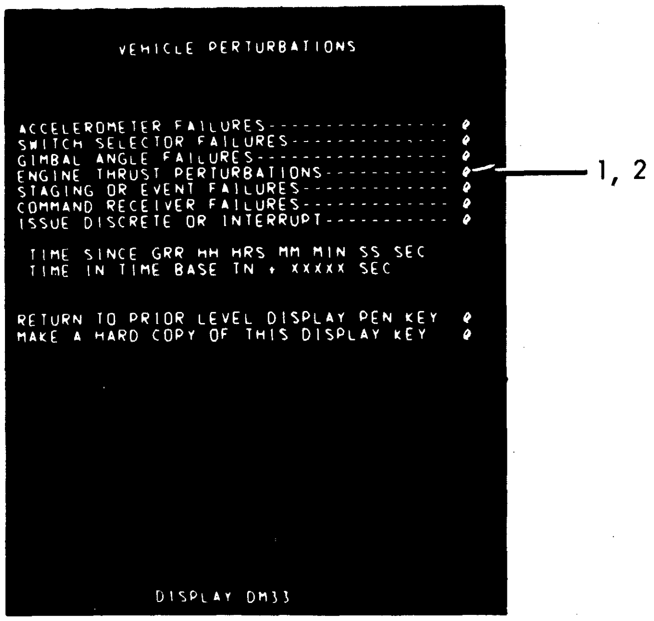
All the pertinent information flowing through the real-time data buffer is collected and saved on tape (the 'Post-Processor tape') for later analysis. If the flight programmer has requested SNAPS and TRACES of actual Flight Computer memory locations during



```
    FLIGHT SIMULATION INITIALIZE


    TO SELECT AN OPTION TOUCH LIGHT
    PEN TO APPROPRIATE KEY SYMBOL

        OPTION                             KEY
        ------                             ---

FLIGHT SIMULATION SYSTEM DESCRIPTION-- ●
CYCLE UP INSTRUCTIONS----------------- ●
SELECT/RECORD RUN OPTIONS ------------ ●
ACCESS/MODIFY MOD 44 PROGRAM---------- ●
LOAD/MODIFY/ACCESS LVDC PROGRAM ------ ●
SPECIFY DCS COMMANDS------------------ ●
SPECIFY CIU MEASUREMENTS-------------- ●
SPECIFY OUTPUT REQUIREMENTS----------- ●
SPECIFY FLIGHT PAUSE POINTS----------- ●
SPECIFY SIMULATED FLIGHT MODE -------- ●
SPECIFY FORTRAN MODEL ---------------- ●
SPECIFY VEHICLE MODEL ---------------- ●
DEFINE VEHICLE PERTURBATIONS --------- ●
LOAD FORTRAN AND VEHICLE MODEL ------- ●


RETURN TO MAIN OPTIONS---------------- ●
MAKE A HARD COPY OF THIS DISPLAY------ ●



            DISPLAY DM01
```

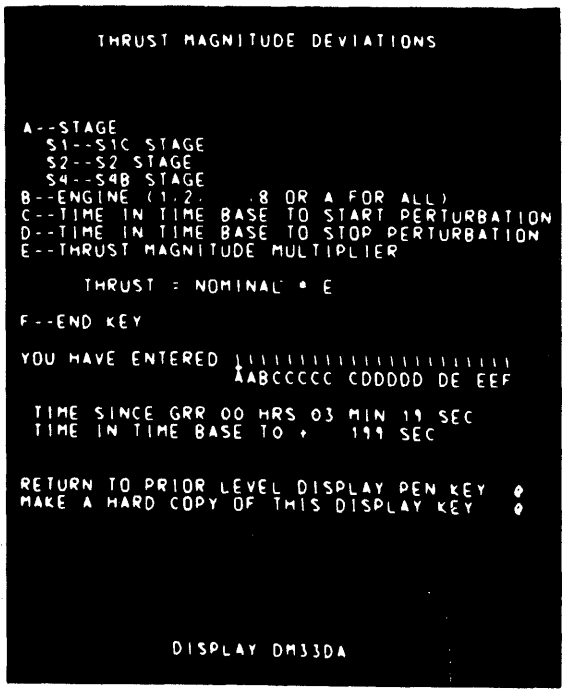
12b

Figure 12—Tutorial displays



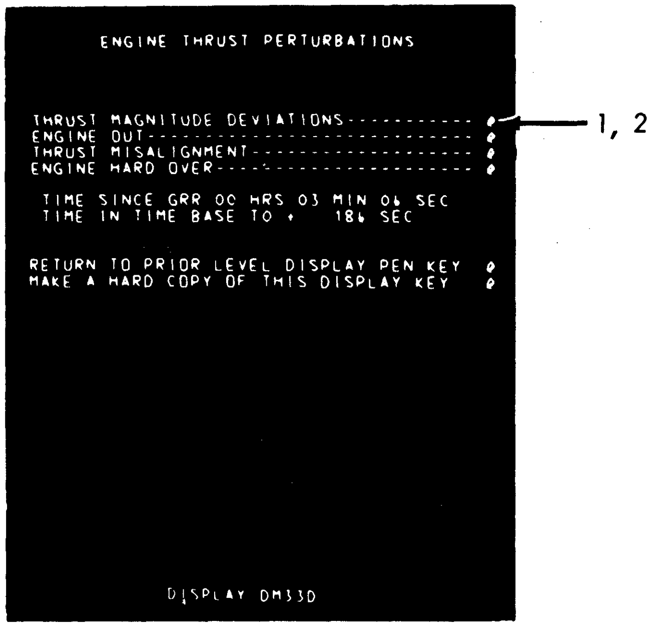
```
              VEHICLE PERTURBATIONS


ACCELEROMETER FAILURES----------------- ◆
SWITCH SELECTOR FAILURES--------------- ◆
GIMBAL ANGLE FAILURES------------------ ◆
ENGINE THRUST PERTURBATIONS------------ ◆
STAGING OR EVENT FAILURES-------------- ◆
COMMAND RECEIVER FAILURES-------------- ◆
ISSUE DISCRETE OR INTERRUPT------------ ◆

 TIME SINCE GRR HH HRS MM MIN SS SEC
 TIME IN TIME BASE TN + XXXXX SEC


RETURN TO PRIOR LEVEL DISPLAY PEN KEY   ◆
MAKE A HARD COPY OF THIS DISPLAY KEY    ◆




                DISPLAY DM33
```

12c



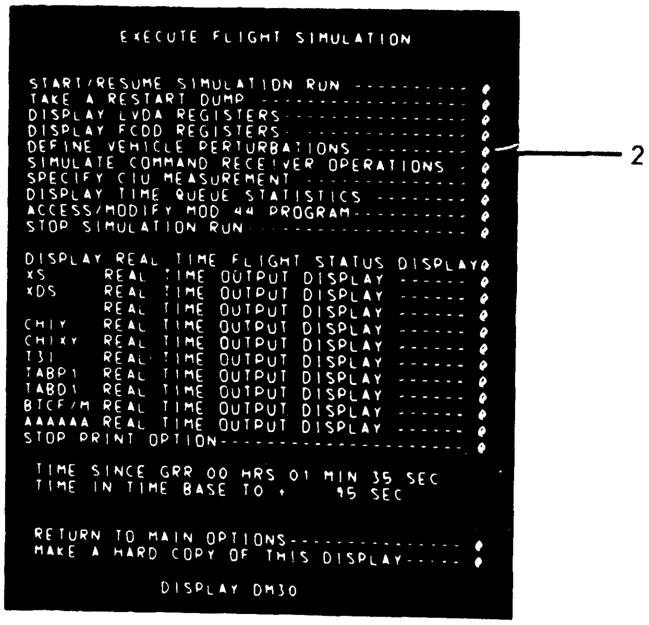
```
        THRUST MAGNITUDE DEVIATIONS


A--STAGE
  S1--S1C STAGE
  S2--S2 STAGE
  S4--S4B STAGE
B--ENGINE (1,2,    .8 OR A FOR ALL)
C--TIME IN TIME BASE TO START PERTURBATION
D--TIME IN TIME BASE TO STOP PERTURBATION
E--THRUST MAGNITUDE MULTIPLIER

     THRUST = NOMINAL * E

F--END KEY

YOU HAVE ENTERED ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
                 AABCCCCC CDDDDD DE EEF

 TIME SINCE GRR 00 HRS 03 MIN 19 SEC
 TIME IN TIME BASE TO +    199 SEC


RETURN TO PRIOR LEVEL DISPLAY PEN KEY   ◆
MAKE A HARD COPY OF THIS DISPLAY KEY    ◆



              DISPLAY DM33DA
```

12e



```
          ENGINE THRUST PERTURBATIONS


  THRUST MAGNITUDE DEVIATIONS----------- ◆
  ENGINE OUT---------------------------- ◆
  THRUST MISALIGNMENT------------------- ◆
  ENGINE HARD OVER---------------------- ◆

   TIME SINCE GRR 00 HRS 03 MIN 06 SEC
   TIME IN TIME BASE TO +    186 SEC


  RETURN TO PRIOR LEVEL DISPLAY PEN KEY  ◆
  MAKE A HARD COPY OF THIS DISPLAY KEY   ◆




                 DISPLAY DM33D
```

12d



```
          EXECUTE FLIGHT SIMULATION

START/RESUME SIMULATION RUN ------------ ◆
TAKE A RESTART DUMP -------------------- ◆
DISPLAY LVDA REGISTERS------------------ ◆
DISPLAY FCDD REGISTERS------------------ ◆
DEFINE VEHICLE PERTURBATIONS ----------- ◆
SIMULATE COMMAND RECEIVER OPERATIONS  -  ◆
SPECIFY CIU MEASUREMENT ---------------- ◆
DISPLAY TIME QUEUE STATISTICS ---------- ◆
ACCESS/MODIFY MOD 44 PROGRAM------------ ◆
STOP SIMULATION RUN -------------------- ◆

DISPLAY REAL TIME FLIGHT STATUS DISPLAY◆
XS      REAL TIME OUTPUT DISPLAY ------- ◆
XDS     REAL TIME OUTPUT DISPLAY ------- ◆
        REAL TIME OUTPUT DISPLAY ------- ◆
CHIY    REAL TIME OUTPUT DISPLAY ------- ◆
CHIXY   REAL TIME OUTPUT DISPLAY ------- ◆
T31     REAL TIME OUTPUT DISPLAY ------- ◆
TABP1   REAL TIME OUTPUT DISPLAY ------- ◆
TABD1   REAL TIME OUTPUT DISPLAY ------- ◆
BTCF/M  REAL TIME OUTPUT DISPLAY ------- ◆
AAAAAA  REAL TIME OUTPUT DISPLAY ------- ◆
STOP PRINT OPTION----------------------- ◆

 TIME SINCE GRR 00 HRS 01 MIN 35 SEC
 TIME IN TIME BASE TO +    95 SEC

 RETURN TO MAIN OPTIONS----------------- ◆
 MAKE A HARD COPY OF THIS DISPLAY------- ◆

              DISPLAY DM30
```

12f

instruction execution, this information is saved on the Post-Processor tape. The Post-Processor tape may be processed immediately or at a later date.

When processing the tape, the user has several options available to him through use of the display system. He may selectively dump any data on the tape and request that the data be converted to decimal form in specified units prior to printing. He may have his data printed in a tabular form or he may plot data on the display unit. Special calculations may be performed on some of the data and the results printed or displayed. He may print or plot errors between various quantities to verify that the flight program results agree with the 6-DOF simulator.

## CONCLUSION

The problem of designing an on-line software checkout facility and ensuring that the programmer/engineer has the capabilties he requires is a complex task. The techniques of simulation, the selection of equipment, and the methods employed for man-computer and computer-computer interface must be carefully weighed. The requirement for pin-point accuracy in the Laboratory resulted in a real-time multiprogrammed system which is proving an invaluable tool for assisting in the development and checkout of the flight programs. It has made possible the development of flight software which can be relied upon to a much greater extent than before and has reduced the amount of time necessary to produce it. In effect, the Laboratory provides the necessary aids toward producing successful flight software.

Some of the software concepts employed in the Laboratory may certainly be applied in related areas of simulation technology. The operating system and display support software have direct conceptual application in airborne and space vehicle simulators.

## BIBLIOGRAPHY

1 L J CAREY W A STURM
*Space software: At the crossroads*
Space/Aeronautics Vol 50 No 7 1968 62-69

2 J L GROSS
*Real time hardware-in-the-loop simulation verifies performance of Gemini computer and operational program*
Simulation Vol 9 No 3 1967 141-148

3 E C VAN HORN
*Three criteria for designing computing systems to facilitate debugging*
Communications 1968 II-5 360-365

4 T H WITZEL J S HUGHES
*Flight software development laboratory*
IBM Corp Huntsville Ala IBM No 68-U60-0022

5 H WYLE G J BURNETT
*Management of periodic operations in a real-time computation system*
Proc FJCC 1967 201-208

6 E YOURDON editor
*Real time systems design*
Information and Systems Institute Cambridge Mass 1967

7 L L ZIMMERMAN
*On-line program debugging—a graphic approach*
Computers and Automation 1967 16-11 30-34

8 *Astrionics system handbook*
Marshall Space Flight Center Huntsville Ala MSFC No IV-4-401-1

9 *LVDC equation defining document for the Saturn V flight programs*
Marshall Space Flight Center Huntsville Ala MSFC No III-4-423-15

# A hybrid frequency response technique and its application to aircraft flight flutter testing

*by* J. M. SIMMONS, J. W. BENSON
and J. P. FIEDLER

*Lockheed-Georgia Company*
Marietta, Georgia

## INTRODUCTION

Large aircraft, such as the Lockheed C-5A, can be forced to resonate on the ground in a large number of closely coupled vibration modes which involve the combined motion of lifting and control surfaces, fuselage and engines. During flight, atmospheric disturbances can also excite these vibrational resonances, though, under normal conditions, they are damped to a safe level because the airstream is able to extract energy from the vibrating structure. However, there exists the aeroelastic phenomenon called flutter[1]—under certain conditions the structure is able to extract energy from the airstream and the amplitude of a resonance can very rapidly increase to a destructive level. Clearly, the damping of all resonances must remain positive throughout a wide range of flight conditions. This is verified by flight flutter test programs during which aircraft are proven safe at an airspeed and altitude before proceeding to a higher airspeed. In one method of flutter testing of large aircraft, the resonant modes are excited during flight by oscillatory forces from aerodynamic vanes.[2] A frequency sweep technique is used; the frequency of the oscillatory forces is varied continuously from about 1 to 30 Hz. Accelerometers or other transducers indicate the response at various locations on the aircraft. After an excitation sweep, the frequencies and measures of damping of the resonances are determined, and a decision is made about the safety of a higher airspeed.

The importance of time in a flight flutter test cannot be overemphasized. Data should be gathered in as short a time as possible in order to relieve the problems of high speed, low altitude testing. Furthermore, the analysis should be completed very quickly to minimize non-productive flight time. A successful data reduction system reduces the time-consuming process of record analysis and increases the time available for engineering interpretations and decisions. This has been achieved by using the Lockheed-Georgia Company's hybrid computing system, consisting of four Ci 5000 analog computers interfaced with a CDC 6400 digital computer, and a new rapid frequency response analysis technique.

### Theory of flight flutter testing

The most powerful techniques for system stability analysis have evolved from the study of sets of simultaneous ordinary differential equations.[3] As a result, flutter testing is based on the assumption that the aircraft in flight can be represented as a linear lumped parameter system described by the equations:

$$\begin{aligned}
a_{11}(p)x_1 + a_{12}(p)x_2 + \cdots + a_{1k}(p)x_k &= f_1(t)\\
a_{21}(p)x_1 + a_{22}(p)x_2 + \cdots + a_{2k}(p)x_k &= f_2(t)\\
\cdots\cdots\cdots\cdots\cdots\cdots&\cdots\cdots\\
a_{k1}(p)x_1 + a_{k2}(p)x_2 + \cdots + a_{kk}(p)x_k &= f_k(t)
\end{aligned}$$

$x_k$ are the coordinates of the system, $f_k(t)$ are the exciting forces and $a_{i\ell}(p)$ are quadratic functions of the operator $p = d/dt$. Because linear systems obey the superposition principle, one can, without loss of generality, excite an aircraft at a single point. Thus all inputs, except $f_i(t)$ can be made zero. Using the Laplace transform, transfer functions can be formed between the $f_i$ and $x_j$, having the general form

$$L\left[\frac{x_j(t)}{f_i(t)}\right] = \frac{X_j(s)}{F_i(s)} = \frac{b_m s^m + \cdots + b_1 s + b_0}{a_n s^n + \cdots + a_1 s + a_0} = \frac{N(s)}{D(s)}$$

where s is the transform variable.

Although N(s) may change for several input-output combinations, the denominator polynomial D(s) is the same. D(s) = 0 is the characteristic equation of the system and the location of its roots, $s = \sigma + j\omega$, in the complex plane defines the stability of the system. Each structural resonance corresponds to a conjugate pair of roots. A flight flutter testing technique that is used involves excitation of the aircraft by a sinusoidal force input which is swept slowly from about 1 to 30 Hz. By measuring the frequency response (both amplitude and phase) of each transducer signal $x_j$ to the exciting force $f_i$, and by applying the frequently used technique of Kennedy and Pancu,[4] $\sigma$ and $\omega$ can be found for all lowly damped resonances. The technique of Kennedy and Pancu is based on a relationship between the damping and the rate of change of phase with frequency (of $x_j$ relative to $f_i$) as the frequency sweeps through a resonance. The variations of $\sigma$ and $\omega$ with airspeed and Mach number show the stability trends of the aircraft. For lowly damped resonances, $\sigma \approx -\zeta\omega$, where $\zeta$ is the familiar damping ratio[3] and $\omega$ can be taken as the circular frequency at amplitude resonance. In this paper the emphasis will be not on the theory of flight flutter testing, but on the hybrid technique for measuring the frequency response of the transducer signals $x_j$ to the exciting force $f_i$.

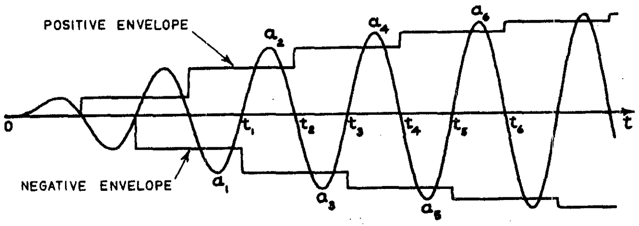


Figure 2—Data signal showing envelopes and zero crossings

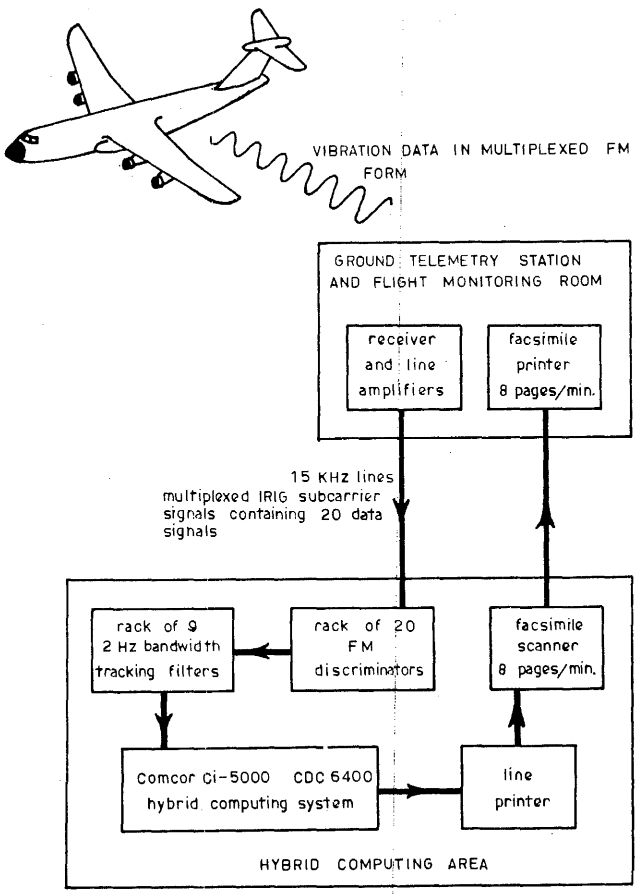

VIBRATION DATA IN MULTIPLEXED FM FORM
GROUND TELEMETRY STATION AND FLIGHT MONITORING ROOM
receiver and line amplifiers
facsimile printer 8 pages/min.
15 KHz lines multiplexed IRIG subcarrier signals containing 20 data signals
rack of 9 2 Hz bandwidth tracking filters
rack of 20 FM discriminators
facsimile scanner 8 pages/min.
Comcor Ci-5000 CDC 6400 hybrid computing system
line printer
HYBRID COMPUTING AREA

Figure 1—The data reduction system

## The hybrid frequency response system

The new data reduction system for the C-5A flight flutter test is shown in Figure 1. Twenty of the data signals, which are telemetered from the aircraft to the ground receiving station, are further transmitted in multiplexed frequency modulated form via 15 KHz lines to the hybrid computing area two miles away. Discriminators restore the twenty signals to

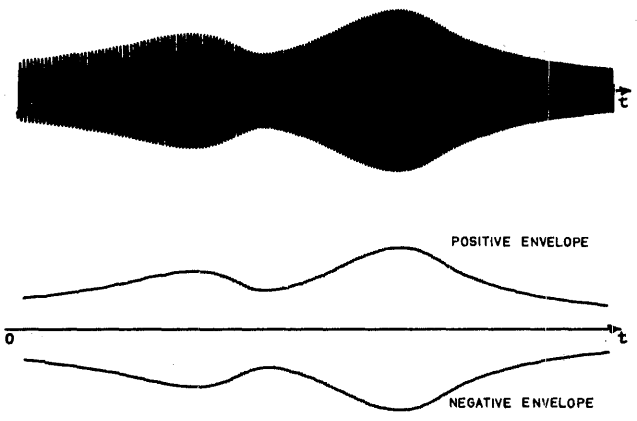


Figure 3—Typical data signal and the computer generated envelopes

analog form and nine of these are selected for further processing. The first operation is bandpass filtering. In each of nine heterodyne-type tracking filters, the center frequency of the 2 Hz passband is continuously tuned to track the excitation frequency sweep. The outputs from the tracking filters oscillate about zero volts and are available in real-time at one of the four Ci 5000 analog computers. These filter outputs become the data signals for input to the hybrid computer and are quite clean sinusoids with slowly varying frequency (Figures 2 and 3). The one excitation signal exhibits approximately constant amplitude but the eight response signals exhibit maxima at the aircraft structural resonances.

## Data compression

The next operation is the storing, in real-time in the central memory of the CDC 6400 digital computer, the times at which all zero-crossings occur, and the peak amplitudes of all cycles. The time of a positive-going zero-crossing, such as $t_3$ in Figure 2 is stored with the amplitude $a_2$ of the previous positive peak. A negative-going zero-crossing time such as $t_4$ is stored with the amplitude $a_3$ of the previous negative peak. This data compression is possible since the outputs of the tracking filters are quite clean sinusoids and more frequent sampling would yield redundant data. If needed for other applications, further data compression could be achieved by discarding some amplitudes and zero-crossing times during parts of the frequency sweep which contain no structural resonances. The necessary real-time digital computing could be performed if the central processor time allocated to the program is greater than the ten percent presently used.

## The envelope detection circuit

The peak amplitudes are generated by applying each data signal to one of the nine envelope detection circuits (Figure 4). Each circuit consists of two nearly identical circuits; one for the positive side and one for the negative side of the data signal. Amplifiers A, B, C, D and E in Figure 4 form a positive envelope circuit utilizing two mode-controlled integrators (D and E) as a track/hold pair, and a first order loop (A, B and C) as a maximum-value circuit. Since the amplifier A represents a perfect diode, the loop acts like a high-gain lag when the input is greater than the output of B and the "diode" A is forward biased. When the input falls below the output of B, the "diode" becomes reverse biased and B is forced to hold at its last value.

As long as the input is positive, comparator output

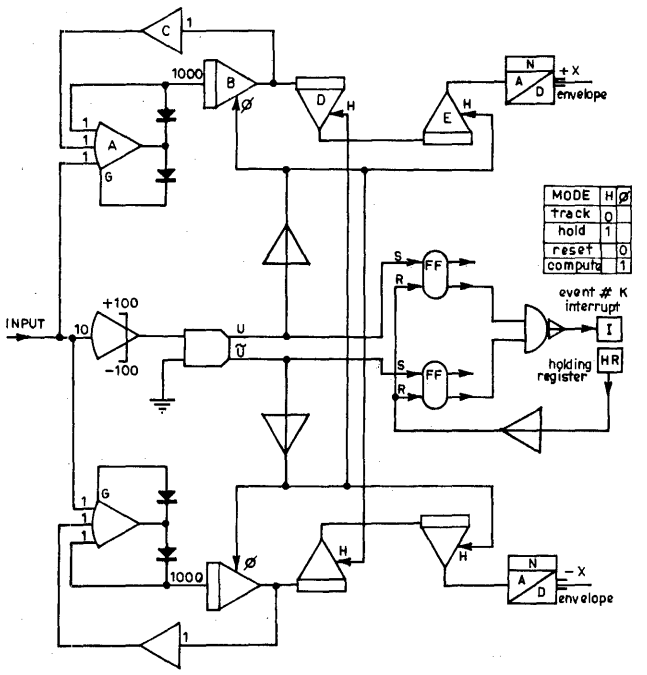


Figure 4—The envelope detection circuit

U is true, B is in the compute mode, D is tracking B, and E is holding the previous peak. As the input goes negative, U goes false, B resets to zero, preparing the maximum-value circuit for the next positive signal, D holds the last voltage from B, and E tracks the new peak from D. Thus, with the positive envelope on E updating on each negative-going zero-crossing of the input, the envelope has staircase-like discontinuities as shown in Figure 2, though it is smoother in practice when more slowly varying frequencies are used. Figure 3 is a segment of typical aircraft data. The second half of each circuit generates the negative envelope in a similar manner (Figure 2). All of these envelope voltages are input continuously to analog-to-digital converters. A practical upper frequency limit to the circuit, using the gains in Figure 4, is 120 Hz. With other gains the useful frequency range could be shifted so that the upper frequency limit is approximately 2000 Hz.

## The hybrid interface and data storage

Each analog console, with its associated interface, contains 32 channels of analog-to-digital conversion. During real-time, when the hybrid system is sensitive to interrupts and peripheral processor patterns (precompiled I/O programs stored within and executed by one of the CDC 6400 peripheral processors), it is possible to transfer nine data words from the sample-and-hold amplifiers, via the analog-to-digital converters, to central memory in less than 300 microseconds.

The actual mechansim of data storage is initiated by leading edge I/O interrupts which cause the previously defined patterns (programs) in the peripheral processor to transfer the data to a temporary buffer in central memory. No central processor time is required for this operation. Upon completion of I/O, the central processor takes the data from the temporary buffer and packs it in an array with four data words per central memory word. Here it is stored until the frequency sweep is completed and the post real-time processing is begun. The leading edge I/O interrupt is activated asynchronously by a positive-going zero-crossing of one of the data channels. This causes the instantaneous digital value of the corresponding positive envelope (from the analog-to-digital converter) to be stored together with the time at which the zero-crossing occurs. Similarly, negative-going zero-crossings set interrupts which initiate the storage of negative-going zero-crossing times and corresponding negative envelope amplitudes.

During the real-time phase, when timing is most critical, the requirements on the central processor are reduced to that of transferring and packing data within central memory. This requires approximately ten percent of the central processor for nine channels at signal frequencies of 30 Hz. Thus the central processor is readily available to service other hybrid programs or batch digital programs as required.

Using central memory only, data for up to 32,000 signal cycles can be stored. It might be possible to increase this number greatly by using the disk file or magnetic tape. However, the Lockheed-Georgia Company's hybrid computing facility is a time-shared system and it was necessary to program this problem for time-sharing compatibility. The system contains a CDC 6638 disk and four CDC 607 tape drives, and is strongly file oriented, using the disk for intermediate file storage during input and output. In a time-critical problem such as flight flutter testing, the disk might not be available for mass data storage, since it can be in use on a non-interruptable channel for several seconds under certain I/O conditions. When real-time mass data storage is required, the 607 tape drives are generally used. These drives can be assigned to a specific problem and can normally be accessed within 500 milliseconds. However, it is necessary to use a central memory buffer capable of storing approximately one second duration of data.

If the restrictions of time-sharing are removed and if the system can be dedicated to flutter testing, the disk, as well as the tapes, becomes available for real-time data storage. The size of the program could be increased even further by using multiple analog consoles and interfaces to achieve parallel data conversion and transmission.

## The hybrid time measurement circuit

A problem arose in the accurate measurement (within ten microseconds) of the time intervals between the interrupts on as many as nine channels.

The problem was complicated by the possibility of virtually simultaneous zero-crossings. Although the digital computer includes a Precision Interval Generator, which downcounts at a rate of 500 KHz, the attempts to use it for accurate timing of events, external to the digital computer, were unsuccessful. This was mainly because of the difficulty of handling simultaneous interrupts and the effect of data-link delays (including software delays and delays, which could be of the order of a millisecond, arising when the computer must finish a previously initiated or

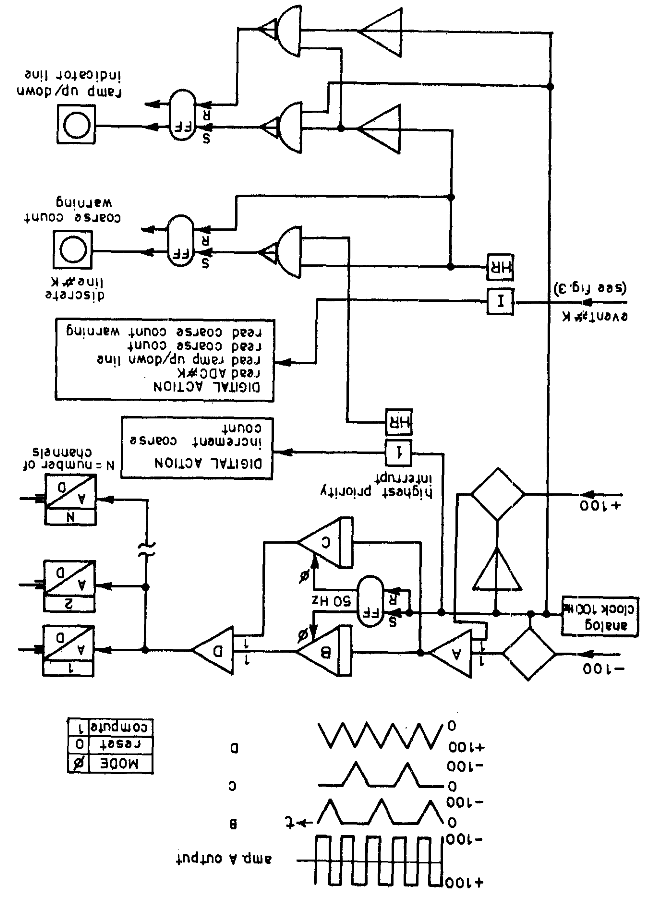

ramp up/down indicator line
coarse count warning
discrete line #K
DIGITAL ACTION read ADC#K read ramp up/down line read coarse count read coarse count warning
DIGITAL ACTION increment coarse count
N = number of channels
highest priority interrupt
event #K (see fig. 3)
+100
analog clock 100 KHz
-100
50 HZ
MODE Ø reset 0 compute 1
amp A output

Figure 5—The time measurement circuit

higher priority task before attending to the next).

These difficulties were avoided by developing a new hybrid technique to take advantage of the sample/hold feature of the analog-to-digital converters (ADC). This time measurement circuit is described with reference to Figure 5. The basis is the generation of a time-synchronized voltage waveform which represents the fine count and is fed into one ADC for each of the data channels to be monitored. The waveform selected is a 0—100—0 volt triangular wave with a 20 millisecond period. It is generated by two complementary integrators controlled by the analog clock which counts down frequencies from a mega-Hertz crystal oscillator. Because one integrator is always in reset while the other is integrating, the synchronization of the output at zero volts is assured at the beginning of each new cycle.

By connecting each holding register (which goes true when an interrupt is set and remains true until the central processor begins action on the interrupt) to the sample/hold controller of the corresponding ADC, this fine count voltage can be held until the digital computer can read it. The coarse count is purely digital and is incremented at the beginning of each cycle by a subroutine which is called from the highest priority interrupt. A logical signal is required if the coarse count is incremented during the time between holding and reading an ADC. This signal is obtained by "ANDing" the holding registers of the coarse count and the zero-crossing interrupts. By using this signal to set a flip-flop which feeds a discrete control line, and by reading this line at the same time as the digital computer reads the ADC, the coarse count portion of the stored time may be decremented if necessary.

The triangular wave is an easily synchronized signal with no discontinuities. For timing purposes, however, it is necessary to know whether the wave is ramping up or down at the time of reading. This is achieved by using, for each channel, a flip-flop tied into a discrete line. The flop-flop normally tracks the analog clock, but maintains the present state when the holding register indicates that the interrupt is in progress. A zero-crossing then triggers an interrupt which simultaneously initiates the digital computer, holds the ADC and the ramp up/down flip-flop, and actuates the gate of the coarse count warning flip-flop. After recognizing the interrupt, the digital computer simply reads the ADC and the two discrete lines and stores their values together with the coarse count. Further action may be postponed until the post-real-time phase.

With a ten volt per millisecond excursion of the analog triangular wave, the ADC's are able to resolve the voltages within 0.1 volts. This is equivalent to a timing accuracy of ten microseconds. Better accuracy could be achieved by balancing the integrators and the ADC's for off-set and drift. Typical results from a system without special balancing are presented in Tables I and II. Identical channels were interrupted simultaneously by an analog clock. Table I contains interrupt times (zero-crossing times) for successive interrupts at 5 Hz while Table II contains similar results for interrupts at 100 Hz. The times of simultaneous events on all channels differ by no more than three microseconds. Also the periods between successive interrupts on any one channel differ by no more than three microseconds. At zero-crossing frequencies as high as 100 Hz, the nine channels of the flight flutter program can be sampled with this same accuracy. It is possible to read all 32 ADC channels at each analog console within one millisecond so that, at a sampling rate of 100 Hz, each interface channel is idle 90 percent of the time. Thus the data frequency or the number of channels could be increased significantly without loss of accuracy.

TABLE I—Successive zero crossing times for five channels interrupted simultaneously at 5 Hz.

| | | | | |
|---|---|---|---|---|
| .969989 | .969990 | .969989 | .969989 | .969991 |
| 1.169992 | 1.169993 | 1.169992 | 1.169995 | 1.169993 |
| 1.369988 | 1.369989 | 1.369988 | 1.369988 | 1.369989 |
| 1.569987 | 1.569988 | 1.569987 | 1.569987 | 1.569988 |
| 1.769992 | 1.769993 | 1.769992 | 1.769992 | 1.769994 |
| 1.969989 | 1.969990 | 1.969989 | 1.969989 | 1.969991 |
| 2.169989 | 2.169989 | 2.169989 | 2.169989 | 2.169989 |
| 2.369992 | 2.369992 | 2.369991 | 2.369991 | 2.369992 |
| 2.569991 | 2.569992 | 2.569991 | 2.569991 | 2.569992 |
| 2.769987 | 2.769987 | 2.769987 | 2.769987 | 2.769989 |
| 2.969991 | 2.969992 | 2.969990 | 2.969990 | 2.969992 |
| 3.169991 | 3.169991 | 3.169991 | 3.169991 | 3.169992 |
| 3.369986 | 3.369987 | 3.369987 | 3.369987 | 3.369989 |
| 3.569989 | 3.569991 | 3.569991 | 3.569989 | 3.569991 |
| 3.769991 | 3.769992 | 3.769991 | 3.769991 | 3.769992 |
| 3.969988 | 3.969989 | 3.969988 | 3.969988 | 3.969989 |
| 4.169989 | 4.169989 | 4.169989 | 4.169989 | 4.169991 |
| 4.369994 | 4.369994 | 4.369994 | 4.369993 | 4.369995 |
| 4.569989 | 4.569989 | 4.569988 | 4.569989 | 4.569991 |
| 4.769989 | 4.769989 | 4.769989 | 4.769989 | 4.769990 |
| 4.969992 | 4.969992 | 4.969993 | 4.969992 | 4.969994 |
| 5.169989 | 5.169991 | 5.169989 | 5.169990 | 5.169992 |
| 5.369988 | 5.369989 | 5.369988 | 5.369987 | 5.369989 |
| 5.569991 | 5.569992 | 5.569991 | 5.569991 | 5.569992 |
| 5.769991 | 5.769991 | 5.769991 | 5.769990 | 5.769991 |
| 5.969987 | 5.969987 | 5.969986 | 5.969986 | 5.969988 |
| 6.169990 | 6.169991 | 6.169991 | 6.169990 | 6.169991 |
| 6.369991 | 6.369992 | 6.369991 | 6.369991 | 6.369992 |
| 6.569988 | 6.569988 | 6.569988 | 6.569987 | 6.569989 |
| 6.769989 | 6.769990 | 6.769990 | 6.769989 | 6.769991 |
| 6.969992 | 6.969992 | 6.969991 | 6.969991 | 6.969993 |
| 7.169989 | 7.169989 | 7.169989 | 7.169989 | 7.169989 |
| 7.369989 | 7.369990 | 7.369989 | 7.369989 | 7.369991 |
| 7.569994 | 7.569994 | 7.569993 | 7.569994 | 7.569995 |
| 7.769989 | 7.769990 | 7.769989 | 7.769989 | 7.769991 |
| 7.969989 | 7.969989 | 7.969989 | 7.969988 | 7.969989 |
| 8.169992 | 8.169992 | 8.169991 | 8.169991 | 8.169993 |
| 8.369990 | 8.369991 | 8.369991 | 8.369990 | 8.369991 |
| 8.569988 | 8.569989 | 8.569989 | 8.569988 | 8.569989 |
| 8.769991 | 8.769991 | 8.769991 | 8.769991 | 8.769992 |

## Post real-time processing

With a maximum lag of only a few cycles after occurrences on the aircraft, a digital description of

TABLE II—Successive zero crossing times for five channels interrupted simultaneously at 100 Hz.

ZERO CROSSING TIMES IN SECONDS

| CHANNEL 1 | CHANNEL 2 | CHANNEL 3 | CHANNEL 4 | CHANNEL 5 |
|---|---|---|---|---|
| .179992 | .179992 | .179992 | .179993 | .179991 |
| .189994 | .189994 | .189994 | .189994 | .189994 |
| .199992 | .199991 | .199992 | .199992 | .199992 |
| .209994 | .209994 | .209994 | .209994 | .209994 |
| .219992 | .219992 | .219992 | .219993 | .219991 |
| .229994 | .229994 | .229994 | .229995 | .229994 |
| .239993 | .239992 | .239992 | .239993 | .239991 |
| .249994 | .249994 | .249994 | .249994 | .249994 |
| .259993 | .259992 | .259992 | .259993 | .259992 |
| .269995 | .269994 | .269994 | .269995 | .269995 |
| .279993 | .279992 | .279992 | .279992 | .279992 |
| .289994 | .289994 | .289994 | .289994 | .289994 |
| .299993 | .299992 | .299992 | .299992 | .299991 |
| .309994 | .309994 | .309994 | .309995 | .309994 |
| .319993 | .319992 | .319992 | .319993 | .319992 |
| .329994 | .329994 | .329994 | .329996 | .329994 |
| .339992 | .339991 | .339992 | .339992 | .339992 |
| .349994 | .349994 | .349994 | .349996 | .349994 |
| .359993 | .359992 | .359993 | .359994 | .359992 |
| .369994 | .369994 | .369994 | .369996 | .369994 |
| .379993 | .379991 | .379992 | .379992 | .379992 |
| .389994 | .389994 | .389994 | .389995 | .389994 |
| .399993 | .399992 | .399992 | .399992 | .399992 |
| .409994 | .409993 | .409994 | .409994 | .409994 |
| .419992 | .419990 | .419991 | .419992 | .419991 |
| .429994 | .429994 | .429994 | .429994 | .429993 |
| .439993 | .439991 | .439992 | .439992 | .439992 |
| .449994 | .449994 | .449994 | .449995 | .449994 |
| .459992 | .459991 | .459992 | .459992 | .459991 |
| .469994 | .469994 | .469994 | .469996 | .469994 |
| .479992 | .479991 | .479992 | .479992 | .479992 |
| .489994 | .489994 | .489994 | .489994 | .489994 |
| .499993 | .499991 | .499992 | .499992 | .499992 |
| .509994 | .509994 | .509994 | .509995 | .509994 |
| .519992 | .519991 | .519992 | .519993 | .519992 |
| .529995 | .529994 | .529994 | .529994 | .529994 |
| .539993 | .539992 | .539992 | .539993 | .539992 |
| .549995 | .549994 | .549994 | .549995 | .549994 |
| .559992 | .559992 | .559993 | .559992 | .559993 |
| .569995 | .569994 | .569994 | .569996 | .569994 |

nine signals can be stored in the CDC 6400. The computer can be ordered to start or stop accepting real-time data either at the console or by remote switches in the flight test monitoring room. After a stop order the stored data is immediately processed. Both signal zero-crossing times and amplitudes undergo conventional digital smoothing. Next an amplitude-versus-frequency history is generated for all nine signals from their peak amplitude values and the time intervals between zero-crossings of the excitation signal. A phase-versus-frequency history of each response signal, relative to the excitation, follows by comparing zero-crossing times in each response with those in the excitation. Thus the hybrid computer is used as a frequency response (or transfer function) analyzer. By searching through the values of the response envelopes, it is able to find the resonances, calculate their frequencies, and normalize their amplitudes by the corresponding amplitudes of the excitation. An increase with airspeed of the normalized amplitude of a resonance can indicate a decrease in its damping and in this way the aircraft stability trends can be followed. If the frequency sweep is sufficiently slow and if the actual forcing of the aircraft is accurately represented by the excitation signal, the computer can use the phase information and the technique of Kennedy and Pancu[4] to separate closely coupled resonances and calculate their damping. Because these post real-time operations involve conventional digital programming, details are irrelevant in this presentation.

Typically, the answers, from eight response signals and a sweep from 1 to 30 Hz lasting 120 seconds, begin to appear on the line printer approximately three seconds after the end of the sweep. For its versatility a facsimile machine is used to transmit copies of the line printer output at eight pages per minute to its remote terminal in the flight test monitoring room. A remote line printer or display scope could quite easily have been used.

## SUMMARY

The hybrid frequency response technique has made possible very rapid data reduction during aircraft flight flutter testing when time saving is extremely important. Previously, such data reduction has been performed in post real-time, to a large extent by hand, from chart recordings. The savings in aircraft flight time, and the increased number of channels which can be analyzed, fully justify the use of a large computer. It is worth comparing this hybrid system with other systems which were considered.

A different approach could be based on the sampling of the data signals at such a high frequency that peak amplitudes and zero-crossing times could be detected digitally, in post real-time, by interpolation between the samples. When the signals are quite clean sinusoids of slowly varying frequency, this method leads to much redundant data and a large storage requirement. Furthermore, it was found that the use of nine data signals and frequencies up to 30 Hz requires that the computer accept a prolonged data input rate far greater than its capability.

Some data compression can be achieved by the use of the Fast Fourier Transform[5] which requires a minimum sampling rate of at least twice the highest frequency of interest.[6] Thus, a sweep from 1 to 30 Hz with 120 seconds duration requires at least 7,200 samples per channel. This is approximately twice the number taken by the hybrid technique. On the CDC 6400 a Fast Fourier Transform of 8192 samples takes approximately sixty seconds per channel using software. This is prohibitively long for flight flutter testing when compared with the three seconds for nine channels taken by the hybrid technique. A Fast Fourier Transform using hardware would be much faster but such a unit was not available. Transform techniques are more applicable to transient and random signals than to slow frequency sweeps.

Separate commercial frequency response analyzers for each data channel could be interfaced with a digital computer through analog-to-digital converters but it is extremely difficult to justify the purchase of a number of such units when a very large hybrid computer is available. Certainly a digital computer is needed to perform the many logical operations which separate the important resonances and discard less important ones. To obtain numerical values of damping, digital operations appear necessary. The hybrid computer has the additional advantage of making possible many convenient forms of system control and display to further aid in saving aircraft flight time.

This data reduction system was developed for use in a flight flutter test program but it should be adaptable to other situations calling for very fast reduction of slow frequency sweeps. The present application requires only one analog console and about 25 percent of the available central memory but it could be expanded to use all four analog consoles and interfaces to give a capability for 40 data channels. Of course, this would dedicate the system. Some elements, such as the hybrid time measurement circuit, could find even wider application.

## ACKNOWLEDGMENTS

We wish to acknowledge the valuable help of M. E. McCoy, A. Roberts, M. Elder and J. Hatley of the Hybrid Computing Department and L. A. Tolve, S. W. Robinson and W. F. Grosser of the Aeromechanics Division. We are also grateful to Mrs. Clara Culpepper for the typing of the manuscript.

## REFERENCES

1 R L BISPLINGHOFF H ASHLEY R L HALFMAN
*Aeroelasticity*
Addison-Wesley Pub Co Inc Cambridge Mass 1955 Chap 1

2 G GRIMM J PHILBRICK
*Flight flutter testing—Recently developed techniques in excitation and data reduction*
IAS Natl Summer Meeting Los Angeles 1960 No 60-91

3 V V SOLODOVNIKOV
*Introduction to the statistical dynamics of automatic control systems*
Dover Pub N Y 1960 Chap 2

4 C C KENNEDY D C P PANCU
*Use of vectors in vibration measurements and analysis*
Journal of Aeronautical Science Vol 14 1947 603-625

5 J W COOLEY J W TUKEY
*An algorithm for the machine calculation of complex fourier series*
Math of Computation Vol 19 1965 297-301

6 C E SHANNON
*Communication in the presence of noise*
Proc IRE Vol 37 No 11 1949

# AMERICAN FEDERATION OF INFORMATION PROCESSING SOCIETIES (AFIPS)

# 1969 FALL JOINT CONFERENCE COMMITTEE

*Chairman*

Jerry L. Koory
Planning Research Corporation

*Vice Chairman*

Ted Braun
Applied Computer Technology Corporation

*Treasurer*

Michael Baran
System Development Corporation

*Secretary*

Robert A. Berman
The RAND Corporation

*Education Program*

Fred Gruenberger, Chairman
San Fernando Valley State College
Don Kehbiel
Santa Monica City College
Roger Mills
TRW Systems Group
Robert White
Informatics, Inc.

*Exhibits*

S. F. Needham, Chairman
TRW Systems
R. D. Blosser
Autonetics
L. J. Bouser
Hewlett Packard
R. A. Burks
Scientific Data Systems
C. R. Cornwell
Lockheed Electronics
P. P. Gehl
Scientific Timesharing Corporation
R. K. Goran
IBM Corporation
M. C. Rogers
TRW Systems
G. M. Sylvester
Lockheed Electronics
E. G. Walsh
California Computer Products

*Ladies' Program*

Ann L. Rataichak, Chairman
IBM Corporation
Mrs. James O. White, Jr.
Mrs. Keith W. Uncapher
Mrs. Fred Gruenberger

*Local Arrangements*

Al Deutsch, Chairman
Informatics, Inc.
Valerie Maitland
The International Data Exchange
Mel Brown
Compata, Inc.
Les Levitan
The International Data Exchange
Tom Schuman
Informatics, Inc.
Stu Shaffer
System Development Corporation
Jim Smith
Naval Undersea R&D Center
Bob White
Informatics, Inc.

*Printing and Mailing*

Ed Chappeleas, Chairman
IBM Corporation
Glenn W. Murray, Vice Chairman
Autonetics
Edith Taggart
IBM Corporation
Alex Connolly

IBM Corporation
Charles Adamo
Philco Ford
Robert L. Koppel
Scientific Data Systems
Howard Gorman
Autonetics
Lora Perkins
Autonetics

*Public Relations*

Robert B. Forest, Chairman
Datamation
Janet Eyler
Datamation
Santo A. Lanzarotto
Scientific Data Systems
Dawn Walker
Dawn Walker Public Relations
Mike Murphy
McGraw Hill
Martha Palubniak
McGraw Hill
Mike Creedman

*Publications and Technical Program*

E. M. Grabbe, Chairman
TRW Systems
Warren E. Meyer, Vice Chairman
System Development Corporation
J. W. Redd
TRW Systems
Jack J. Pariser
Hughes Aircraft Company
Guy H. Dobbs
Isaacs, Dobbs System
John J. Rosati
TRW Systems
Art M. Rosenberg
Informatics, Inc.
Allan N. Wilson
General Dynamics
Charlie D. Coleman
IBM Corporation
Alex Hurwitz
IBM Corporation
Donald W. Gade
Aerospace Corporation
Robert E. Perry
Hughes Aircraft Company
Esker J. Harris
IBM Corporation

*Registration*

Frank F. Jurkovich, Chairman
Applied Computer Technology
Patricia M. Riley, Vice Chairman
TRW Systems
Walter L. Dooley
North American Rockwell
Dixie L. Lopez
Precision Data Systems, Inc.
Phyllis W. Yorg
TRW Systems
Irene E. Matthews
TRW Systems
R. A. Hayes
Hughes Ground Systems Support

*Special Activities Committee*

Smith Dorsey, Chairman
Autonetics
Muriel Gustin
Varian Data Machines
Scott Hillman
Autonetics
Robert McCowan
Scientific Data Systems
Robert Steen
IBM Corporation
Paul Thomas
Autonetics

*Liason*

Harry T. Larson
California Computer Products
H. G. Asmus, AFIPS Headquarters
American Federation of Information
Processing Societies
Richard B. Blue, Sr., ACM
TRW Systems Group
Jerry Baker, SCI
Hughes Aircraft Company
Sei Shohara, IEEE
Scientific Data Systems

# REVIEWERS, PANELISTS, AND SESSION CHAIRMEN

## REVIEWERS

Robert P. Abbott
Chacko T. Abraham
Robert M. Aiken
Richard M. Alden
Roy P. Allen
Edward B. Altman
Saul Amarel
L. D. Amdahl
Juan J. Amodei
Robert H. Anderson
L. V. Anderson
T. C. Anderson
Frank D. Anzelmo
Akio Arakawa
Majid Arbab
Paul Armer
George N. Arnovick
Joel D. Aron
M. M. Astrahan
Pauline A. Atherton
Donald C. Augustin
H. L. Babin
George F. Badger, Jr.
Philip R. Bagley
Jerry H. Baker
N. Addison Ball
Michael Ballot
Robert Balzer
Allen E. Barlow
Ben B. Barnes
Robert M. Barnett
James P. Bartlett
A. Batenburg
Frank Bates
John A. Bayless
W. R. Beam
C. K. Bedient
G. A. Bekey
Robert W. Bemer
R. D. Benham
Russell Bennett
Frank Bequaert
Paul T. Berning
M. I. Bernstein
Paul W. Berthiaume
Lawrence Beruc
William P. Bethke
L. L. Bewley
Claude D. Birkhead
Donald V. Black
James A. Bloomfield
Daniel G. Bobrow
Morris J. Bodoia
Garret Boer
Gordon R. Bolton
Harold Borko
E. I. Bosch
Sherman H. Boyd
A. M. Bradley
Robert D. Brandsberg
Harvey Bratman
E. L. Braun
Barbara Brawn
Robert Brennan
Melvin A. Breuer
Carl N. Brooks
Barry W. Brown
J. Reese Brown, Jr.
G. E. Bryan
Wener Buchholz
T. D. Buettell
Leslie L. Burns
Warren P. Burrell
C. A. Caceres
Myron A. Calhoun
Peter Calingaert
E. David Callender
Thomas W. Calvert
D. J. Campbell
Anthony V. Campi
Rudd H. Canaday
David W. Cardwell
Roy B. Carlson
Robert L. Carmichael
Chester C. Carroll
W. C. Carter
Leonard J. Chaitin
James M. Chambers
Stanley K. Chao
G. G. Chapin
T. E. Cheatham
R. C. Cheek
Li-an L. Chen
J. Chernak
B. F. Cheydleur
G. Chingari
C. K. Chow
W. F. Chow
R. F. Churchhouse
E. H. Clamons
W. Douglas Climenson
Lawrence J. Clingman
A. Ben Clymer
Edward G. Coffman, Jr.
Dan Cohen
Edmund U. Cohler
Walter L. Colby
L. Stephen Coles
Albert H. Coltin
Steve Condon
Thomas J. Condon
Ralph B. Conn
Michael M. Connors
Barbara Conrad
Robert Constant
Alfred E. Corduan
J. L. Corbett
W. A. Cornell
Ira W. Cotton
George A. Coulman
F. C. Cowburn
T. D. Cox
Richard L. Crandall

Arthur J. Critchlow
D. L. Critchlow
James J. Croke
Herbert A. Crosby
Joseph D. Crunkleton
Nicholas Cserhalmi
Charles A. Csuri
Joseph F. Cunningham
Alfred G. Dale
Donald A. Darms
C. M. Davis
Kenton S. Day
Stephen Peter de Jong
Peter B. Denes
Peter J. Denning
Weldon C. Dennis
Karl S. Detzer
U. Clarke S. Dilks
Heinz Dinter
Donald L. Dittberner
George G. Dodd
Richard K. Dove
John C. Duffendack
Michael A. Duggan
John J. Dulin
Arnold I. Dumey
T. J. Dylewski
Lester D. Earnest
Lita B. Edwin
C. A. Eggert
Raymond Eisenstark
Robert F. Elfant
Pete England
Warren J. Erikson
F. Dennis Erwin
Edward R. Estes
S. E. Estes
David L. Evans
Carl C. Farrington, Jr.
George A. Fedde
Julian Feldman
Frank R. Field, Jr.
Robert T. Filep
T. R. Finch
Ray Fitzgerald
James L. Flanagan
E. Gil Flores
L. E. Fogarty
F. H. Fowler
Margaret R. Fox
Amalie J. Frank
W. Donald Frazer

Roy N. Freed
I. F. Freibergs
C. V. Freiman
Paul J. Friedl
Joyce Friedman
James P. Fry
Lewis M. Fulton
Adolf Futterweit
L. Gainen
Rodger L. Gamblin
Sherbie G. Gangwere
Manuel G. Garcia
Reed M. Gardner
Clarence Giese
M. C. Gilliland
Michael M. Gold
David G. Gordon
Jerome J. Gordon
Robert M. Gordon
D. F. Gorman
John A. Gosden
Malcolm H. Gotterer
Alan J. Gradwohl
Alonzo G. Grace, Jr.
M. N. Greenfield
Donald W. Grissinger
George F. Frondin
Gabriel F. Groner
W. Groth
Otto A. Gutwin
Adolfo Guzman
Thomas G. Hagan
Murray J. Haims
John E. S. Hale
Mark I. Halpern
Richard G. Hamlet
Carl Hammer
Frederick M. Haney
A. G. Hanlon
P. J. Hanratty
John W. Harbaugh
Philip A. Harding
Donald R. Haring
Esker J. Harris
J. O. Harrison
Harry P. Hartkemeier
Elbert Hartsfield
R. Dean Hartwick
S. L. Hasin
Theodore F. Hatch, Jr.
Kenneth E. Haughton

Arthur Hausner
Robert M. Have
Lester C. Hazlett
John Heafner
John D. Heightley
Melvin F. Heilweil
Walter A. Helbig
V. E. Henriques
Paul J. Hermann
Bertram Herzog
George E. Heyliger
John H. Hiestand
A. N. Higgins
Richard H. Hill
Leonard Hirsch
Harold M. Hite
Elias H. Hochman
Alistair D. C. Holden
G. L. Hollander
Arthur W. Holt
Robert L. Hooper
James A. Howard
David K. Hsiao
Barbara Huberman
Thomas A. Humphrey
Earl Hunt
Cuthbert C. Hurd
P. J. Hurley
Gilbert P. Hyatt
Manley R. Irwin
Roy A. Ito
Edwin L. Jacks
Albert S. Jackson
Edward A. Jacoby
Leo F. Jarzomb
Ronald Jefferies
Bruce B. Johnson
Edwin G. Johnson
R. E. Johnson
Walter L. Johnson
Edwin R. Jones
Terence G. Jones
Earl C. Joseph
L. E. Justice
Richard Y. Kain
Marvin J. Kaitz
J. F. Kalbach
Ted Kallner
Akira Kasahara
Charles Kellogg
Joseph E. Kernan

C. W. Kessler
Wan-Lin Kiang
Robert E. King
E. S. Kinney
Philip Kiviat
K. E. Knight
Prentiss Knowlton
Manfred Kochen
H. R. Koen, Jr.
Eldo C. Koenig
C. J. Koester
James S. Koford
Igal Kohavi
Ziv Kohavi
Anthony J. Kolk, Jr.
Deena Koniver
John O. Kopf
G. A. Korn
Ladis D. Kovach
R. L. Kuehn
Carl J. Kuehner
J. H. Kuney
Jerome Kurtzberg
Kenneth C. Kwan
Dominic A. Laiti
Butler W. Lampoon
Daniel J. Lasser
P. Lazarus h
Eric G. A. Lean
Richard C. T. Lee
Y. C. Lee
M. Lehman
John Lennie h
A. S. Lett h
William E. Lewis
W. Wayne Lichtenberger
Hans P. Lie
Leonard R. Lindenmeyer
Carroll R. Lindholm
Robert K. Lindsay
Robert N. Linebarger
Thomas P. Linville
Ho-Nien Liu
Kenneth M. Lochner, Jr.
R. D. Lohman
Henry A. Long
Fred Luconi h
David K. Lynn
Malcolm Macaulay
B. E. F. Macefield
Walter Main
C. M. Malone h

Carl W. Malstrom
Richard L. Mandell h
Michael Marcotty
John Markus
M. E. Maron h
Irvin Marshall
William L. Martin
R. L. Mattison
Harold E. Maruer
Lynn H. Maxson
C. Hugh Mays
M. E. McCoy
Andrew J. McGill
J. L. McKenney
P. T. McKiernan
John McLeod
M. W. McMurran
H. W. Mergler
Michael J. Merritt
Gene S. Metsker
Charles S. Meyer
James C. Michener
Bart J. Michielsen
Kenneth L. Miller
Stephen W. Miller
W. F. Miller
Jack Minker
Gerald Minton
Baker A. Mitchell
E. E. L. Mitchell
Gordon S. Mitchell
Benjamin Mittman
Owen R. Mock
Marion F. Moon
Dana W. Moore
Richard Kelly Moore
Richard A. Moran
Stanley M. Morris
George J. Moshos
Robert A. Mosier
John H. Munson
John K. Munson
Anthony W. Muoio
John J. Murray
F. W. Murray
Robert P. Myers
Jan A. Narud
David Nee
Gary W. Nelson
Richard A. Nesbit
Peter G. Neumann
Allen Newell

Malcolm C. Newey
Fred Newman
William M. Newman
C. B. Newport
R. V. Niedrauer
Norman R. Nielsen
Nils J. Nilsson
N. Nisenoff
Samuel Nissim
J. D. Noe
Ronald A. Nolby
Paul Northrop
William A. Notz
D. R. O'Bell
Joseph A. O'Brien
A. Ockene
Cedric F. O'Donnell
Ken O'Flaherty
John T. O'Neil, Jr.
Lubomyr S. Onyshkevych
G. Oppenheimer
Richard H. Orenstein
Elmer Edwin Osborne
J. T. Owens
Thomas F. Penderghast
Lysel H. Peterson
J. G. Petitt
James K. Picciano
Melvin W. Pirtle
Warren J. Plath
A. V. Pohm
Robert V. Pole
James M. Pomerene
Sigmund N. Porter
John A. Postley
A. W. Potts
L. K. Pounds
M. J. D. Powell
Rebecca C. Prather
R. J. Preiss
J. Paul Pritchard, Jr.
I. C. Pyle
Jesse T. Quatse
James S. Raby
C. V. Ramamoorthy
Bertram Raphael
M. D. Rapkin
A. Karl Rapp
Louis C. Ray
Stanley G. Reed
Harry C. Reinstein
Irwin Remson

William T. Rhoades
Phyllis A. Richmond
Frank C. Rieman
Joseph W. Rigney
Frank D. Risko
Lawrence G. Roberts
R. W. Roberts
D. E. Robison
Nathaniel Rochester
Alan E. Rogers
R. M. Rojko
Michael W. Rolund
Jack Roseman
C. A. Rosen
Morton Rosenberg
Jack L. Rosenfield
Robert R. Rosin
William Edward Ross
R. E. Roundtree, Jr.
Raymond J. Rubey
Paul M. Rubin
Morris Rubinoff
Seymour Z. Rubenstein
Fred Ruffing
Edward C. Russell, Jr.
Roy L. Russo
Jerome D. Sable
Tak Saisho
Erik Salbu
Gerard Salton
John M. Salzer
P. I. Sampath
Jere L. Sanborn
Wendell Sander
F. J. Sansom
Lawrence Sashkin
Helmut M. Sassenfeld
E. S. Savas
Don Savitt
David B. Saylors
W. E. Schiesser
Arthur J. Schneider
Larry C. Schooley
Ernest J. Schubert
Melvyn H. Schwartz
J. E. Schwenker
Sally Y. Sedelow
Thomas K. Seehuus
Warren D. Seider
Robert H. Selzer
Arnold B. Shafritz
David Shansky
Elmer B. Shapiro
Jacke E. Shemer
Paul C. Sheretz
Jerome S. Shipman
Richard R. Shively
Sei Shohara
Paul N. Sholtz
Gerald E. Short
Richard L. Shuey
George T. Shuster, Jr.
Edgar H. Sibley
Roland Silver
Leonard C. Silvern
Q. W. Simkins
R. Simmons
W. D. Simpson
K. D. Sirakides
Patrick G. Skelly
R. A. Slater
W. U. Slauk
Donald R. Slutz
Terry A. Smay
Bernard L. Smith
Kenneth Creston Smith
Leland Smith
Richard V. Smith
L. A. Smitzer
E. W. Snyder
Terry R. Snyder
Gerald N. Soma
L. M. Spandorfer
Charles F. Spitzer
F. W. Springe
Thomas B. Steel, Jr.
Howard H. Steenbergen
John K. Stephens
David H. Stewart
A. J. Stone
Harold S. Stone
Jon C. Strauss
Walter A. Sturm
Maurice E. Suhre, Jr.
Roger K. Summit
William R. Sutherland
R. Taylor
Arthur Teplitz
Larry Tesler
Alan L. Tharp
R. E. Thoman
E. M. Thomas
Gregory L. Thomas
Martin D. Thompson
W. P. Timlake
August A. Toda
Fred M. Tonge
Douglas M. Towne
George R. Trimble, Jr.
Thomas D. Truitt
H. S. Tsou
Frank Tung
G. H. Turner, Jr.
George J. Turner
G. T. Uber
Leonard Uhr
Erwin A. Ulbrich, Jr.
Man T. Ung
William R. Uttal
Richard L. Van Horn
Richard L. Van Tilburg
R. Vichnevetsky
Sam S. Viglione
R. Von Buelow
Alfred H. Vorhaus
Sigurd Waaben
Sven E. Wahlstrom
John V. Wait
R. V. Wakerling
P. Duane Walker
John B. Wallace
Charles J. Walter
C. A. Walton
Ben C. Wang
Gary Y. Wang
Robert L. Ward
I. A. Warheit
Homer R. Warner
M. Cameron Watson
C. W. Watt
Vance Weaver
M. N. Weindling
Leonhard H. Weiner
Ralph R. Wheeler
J. J. Whelan
Malcolm E. White
Gio Widerhold
Jerome B. Wiener
Ronald L. Wigington
Roger C. Wilborn
Lyle C. Wilcox
M. Wildmann
Donald A. Willard
Theodore J. Williams
Thomas G. Williams

Carrel A. Wilson
Nels Winkless
Howard Wishner
Eric W. Wolf
James E. Wolle
John W. Womack
Roger C. Wood
Franz Worth
J. H. Worthington
J. Howard Wright
Kendall R. Wright
Ronald E. Wyllys
J. C. Wyman
J. W. Young
Lawrence S. Young
Daniel C. Zatyko
Norman S. Zimbel
Stuart Zimmerman
Arthur S. Zukin

## PANELISTS

Lynn Abbott
Paul Armer
L. A. Avanzino
Robert Barnett
Robert W. Bemer
Sergio Bernstein
Paul Berthiaume
Melvin Breuer
Alan R. Butcher
James C. Castle
Ken Charshaf
Robert L. Chartrand
R. K. Chooljian
A. Ben Clymer
Aaron H. Coleman
Robert S. Cope
Alex d'Agapeyeff
Roy Davis
William E. DeLair
Ben Erdman
L. E. Fogarty
Les Goldberg
G. R. J. Grosch
Alexander C. Grove
Stanley D. Halper
John W. Hamblen
Peter P. Harris
Joseph O. Harrison
Alan Hecht
Elias H. Hochman
Bernard C. Hogan
Joseph Hootman
John F. Horty
Robert Jefferson
Stephen J. Kahne
Dan D. Kassan
Al J. Knite
R. C. Leader
George F. J. Lelaner
Roger Levien
Robert W. Lucky
Tony Lumpkin
Carl W. Malstrom
Michael Marcotty
John Mayne
C. W. Medlock
Mortimer Mendelsohn
K. Stephen Menger
Paul Metzelaar
Edward E. L. Mitchell
Jack E. Myers
Thomas J. McConnel, Jr.
William H. McKeeman
D. C. McElroy
Carl E. Nelson
K. Okashima
K. Otten
Max Palevsky
Thomas M. Rees
John S. Saloma III
Phillip L. Schiedermayer
Kenneth Schurr
Robert J. Seidel
John D. Seiley
John P. Singleton
Thomas B. Steel, Jr.
Howard Steenbergen
Julius T. Tou
John V. Tunney
Lawrence Urdang
Willis Ware
Milton Warshawsky
Lawrence Weed
John Willner
Joseph H. Wimbrow
Eric W. Wolf
William L. Wooley

## SESSION CHAIRMEN

Morton I. Bernstein
Leon Blitzer
David H. Brandin
Robert R. Brown
Walter Brunner
James Burrows
Michael P. Burwen
Paul S. Collins
Francis L. Goff
Malcolm H. Gotterer
Martin Greenberger
A. H. Halpin
Robert V. Head
Richard Johns
Kenneth W. Kolence
Walter F. Kosonocky
Roy L. Lawrence
Don Lebell
Arthur H. Lipton
Jerome Lobel
William L. Martin
Donald A. Meier
Robert McClure
Bret Nebel
Louis Robinson
Joseph W. Smith
Merlin G. Smith
Robert Stuckelman
Robert L. Thaler

## BEST PRESENTATION AWARD PANEL

Robert H. Glaser
IBM Corporation

James H. Bennett
Applied Logic Corporation

Harry T. Larson
California Computer Products

J. W. Redd
TRW Systems

Rex Rice
Fairchild Semiconductor

## PRIZE PAPER COMMITTEE MEMBERS

# 1969 FJCC LIST OF EXHIBITORS

Access Systems, Inc.
Adage, Inc.
Addison-Wesley Publishing Company, Inc.
Addressograph Multigraph Corporation
Advanced Programming, Inc.
Advanced Systems Inc.
Advanced Terminals, Inc.
AFIPS Press
Airoyal Manufacturing Company
AL/COM Time Sharing Network
Allen Babcock Computing, Inc.
Allied Computer Technology, Inc./Heuristic Systems Division
Alphameric Data Corporation
American Data Systems
American Telephone & Telegraph Company
AMP, Inc.
Ampex Corporation
Anderson Jacobson, Inc.
APL Computing Services
Applied Data Research, Inc.
Applied Digital Data Systems
Applied Dynamics, Inc.
Applied Magnetics Corporation
Applied Peripheral Systems, Inc.
Association for Computing Machinery
Astrocom Corporation
Astrodata, Inc.
Astrosystems, Inc.
Atlantic Research Corporation, Telecommunication Products Dept.
Atlantic Technology Corporation
Audio Devices, Inc.
Auerbach Associates
Auerbach Corporation
Auerbach Info,/Inc.
Automated Business Systems, OEM Products—Div. of Litton Industries
Auto-trol Corporation

Badger Meter Manufacturing—Systems Div.
Beckman Instruments, Inc.
Beehive Electrotech, Inc.
Bell & Howell, CEC/Data Instruments Div.
Beltronix Systems, Inc.
Berkeley Computer Corporation
Beta Instrument Corporation
BIT, Inc.
Boole & Babbage, Inc.
Brogan Associates, Inc.
Bryant Computer Products
Bucode Inc.
Bunker-Ramo Corporation, Business & Industry Div.
Business Automation

Caelus Data Products
California Computer Products, Inc. (CALCOMP)
California Data Systems Corporation
Call-A-Computer, Inc.
Cambridge Memories, Inc.
Century Data Systems, Inc.
Certex, Inc.
Cipher Data Products, Inc.
Clary Datacomp Systems
Clevite Corporation
Codex Corporation
Cogar Corporation—Technology Div.
Cognitronics Corporation
Collins Radio Copmany
Colorado Instruments, Inc.
Comcet, Inc.
Communitype Corporation
Compat Corporation
Compiler Systems, Inc.
CompuSys Inc.
Computek, Inc.
Computer Access Systems, Inc.
Computer Applications Inc.
Computer Automation, Inc.
Computer Communications, Inc.
Computer Corporation of America—Houston, Texas
Computer Design
Computer Displays Inc.
Computer Equipment Corporation
Computer Industries Inc.
Computer Learning Corporation
Computer Link Corporation
Computer Micro-Data Systems, Inc.
Computer Peripherals Corporation
Computer Sciences Corporation
Computer Synetics Inc.
Computer Terminal Corporation
Computer Terminals, Inc.
Computer Test Corporation

Computer Time-Sharing Corporation
Computer Transceiver Systems, Inc.
Computerworld
Consolidated Computer Services
Consolidated Software, Inc.
Control Data Corporation
Courier Terminal Systems, Inc.
Cummins-Chicago Corporation
Cybernetics International
Cytek Information Systems Corporation

Daedalus Computer Products Inc.
DASA Corporation—Data Products Div.
Data Action Corporation
Data Card Corporation
Data Communications Systems, Inc.
Data Computer Systems
Datacraft Corporation
Data Disc, Inc.
Data General Corporation
Data-Link Corporation
Datamark, Inc.
DataMate Computer Systems, Inc.
Datamation
Data Packaging Corporation
Data Power, Inc.
Data Printer Corporation
Data Processing Magazine
Data Products Corporation
Data Products News
Dataram Corporation
Datascan, Inc.
Data Systems News
Data Technology
Datatype Corporation
DataWest Corporation
Datel Corporation
Datran Corporation
Datron Systems Inc.
Delta Data Systems Corporation
DI/AN Controls, Inc.
Digital Development Corporation
Digital Equipment Corporation
Digital Information Devices
Digital Scientific Corporation
Digitronics Corporation
DSI Systems, Inc.
Dura, Div. Intercontinental Systems, Inc.
Dynatronics Operation, Electronics Div., General Dynamics
Dynelec Systems Corporation

Eastman Kodak Company, Business Systems Markets Div.
EDP Central, Inc.
Edutronics, Inc.
Edwin Industries Corporation
EECO
EG & G, Inc., Systems Development Div.
E-H Research Laboratories, Inc.
Electronic Associates, Inc.
Electronic Memories & Magnetics Corporation
Electronic News
EMR Computer
Executive Computer Systems, Inc.

Fabri-Tek, Inc.
Facit-Odhner, Inc.
Factsystem, Inc.
Ferranti Electric, Inc.
Ferroscube Corporation
Ford Industries, Inc.
Foto-Mem, Inc.
Fujitsu Limited

General Automation, Inc.
General Computers, Inc.
GDI Inc.
General Electric Company
  Information Devices
    Bull Corporation
    Custom Power Equipment
    Communication Products
    Information Systems
General Instrument Corporation, Magnehead Div.
General Kinetics Inc.
Gerber Scientific Instruments Company
Graphic Data, Inc.
Graphic Displays Limited (An Electrautom Company)
GRI Computer Corporation

Hayden Publishing Company
Mendrix Electronics
Hewlett-Packard Company
H. F. Image Systems, Inc.
Honeywell—Computer Control Div.
      EDP
Houston Instrucment Div. of Bausch & Lomb
Hypertech Corporation

IBM Corporation
IKOR, Inc.
Imlac Corporation

Inforex, Inc.
Information Control Corporation
Information Data Systems, Inc.
Information Displays, Inc.
Information International
Information Research Associates, Inc.
Information Storage Systems, Inc.
Information Technology & Systems, Inc.
Information Technology, Inc.
Infotec, Inc.
Infotran, Inc.
Institute of Electrical & Electronics Engineers
InterAccess Corporation
Interdata, Inc.
Interface Mechanisms, Inc.
Intermac Corporation
International Communications Corporation
International Computer Products, Inc.
International Data Sciences, Inc.
International Timesharing Corporation
Interplex Corporation
Invac Corporation
Iomec, Inc.

Jacobi Systems Corporation
Jonker Corporation

Kennedy Company
Keyboard Training, Inc.
Keymatic Data Systems Corporation
Kleinschmidt Div. of SCM Corporation
KYBE Corporation
KYBE Corporation

Lambda Electronics Corporation
Leasco Computer, Inc.
Leasco Systems & Research Corporation
Computer Service Group
Lenkurt Electric Company, Inc.
Licon Div. of Illinois Tool Works, Inc.
Linnell Electronics, Inc.
Lipps . . Inc.
Litton Datalog, Div. of Litton Industires
Lockheed Electronics Company—Data Products Div
Logic Corporation

The MacMillan Company
Magnafile, Inc.
Magnusonic Devices Inc.
MAI Equipment Corporation
Marshall Communications
Marshall Data Systems

Mastech Computer Systems, Inc.
Mech-Tronics Corporation
Memorex Corporation
Memory Technology Inc.
Merlin Systems Corporation
Microform Data Systems, Inc.
Microswitch, A Div. of Honeywell
Micro Systems Inc.
Milgo Electronic Corporation
Miller-Ellis Computer Systems, Inc.
Mini-Comp, Inc.
3M Company, Computer Graphics
Modern Data
Mohawk Data Sciences Corporation
Monitor Systems an Aydin Company
Motorola Instrumentation & Control, Inc.
McGraw-Hill Book Company

The National Cash Register Company
The National Cash Register Compnay—Industrial Products Div.
Newell Industries, Inc.
Nissei Sangyo America, LTD.
Nortec Computer Devices Inc.
Novar Corporation
Novation, Inc.

Olivetti Underwood Corporation
Omnitec Corporation, A Div. of Nytronics, Inc.
Oneida Electronics, Inc.

Path Computer Equipment, Inc.
Penril Data Communications, Inc.
Peripheral Data Machines Inc.
Peripheral Equipment Corporation
Philco-Ford Corporation—Western Laboratories Div.
Photon, Inc.
Polaroid Corporation
Potter Instrument Company, Inc.
Prentice Hall, Inc.
Princeton Electronic Products, Inc.

RCA Graphic Systems Div.
RCA Memory Products Div.
Raytheon Company
Raytheon Company, Raytheon Computer Operation
Recortec, Inc.
Redcor Corporation
Remcom Systems, Inc.
Remex Electronics
RFL Industries
Rixon Electronics

Rolm Corporation
Royco Instruments, Inc.

Sanders Associates, Inc.
Sangamo Electric Comapny
Scan-Data Corporation
Scientific Control Corporation
Scientific Data Systems
Scientific Resources Corporation
The Service Bureau Corporation
Shepard Div./Vogue Instrument Corporation
Simulators, Inc.
Singer, Advanced Technology
Singer—Tele-Signal Corporation
Spartan Books
Spiras Systems, Inc, (formerly I.R.A. Systems, Inc)
Stromberg Datagraphix, Inc.
Sy cor, Inc.
Sykes Datatronics, Inc.
Syner-Data,, Inc.
Synergi stics Inc.
SystemInteraction Corporation
Systems Concepts, Inc.
Systems Engineering Laboratories

Tally Corporation
TEC, Inc.
Technitrend, Inc.
Tektronix, Inc.
Telematics, Inc.
Teletype Corporation
Telex Computer Products Div.
Tel-Tech Corporation
Tempo Computers Inc.
Terminal Equipment Corporation
Texas Instruments (Industrial & Government Products Divs.)
Time/Data Corporation
Time Share Peripherals Corporation
Time-Sharing Terminals Inc.
Toko N.Y., Inc.

Tracor Computing Corporation
Transducer Systems, Inc.
Transmission Measurements Inc.
Tri-Data Corporation
Tyco Laboratories, Inc., Digital Devices Div.
Ty-Core, Inc.
Tymshare, Inc.
Typagraph Corporation

Ultronic Systems Corporation
United Computing Systems, Inc.
UNIVAC, Div. of Sperry Rand Corporation
Universal Data Acquisition Company, Inc.
Universal Drafting Machine Corporation
Universal Systems, Inc.
U. S. Time-Sharing, Inc.
UTE (United Telecontrol Electronics)

Vanguard Data Systmes
Varian Data Machines
Vermont Research Corporation
Viatron Computer Systems Corporation
Victor Comptometer Corporation
Video Systems Corporation
Virginia Panel Corporation

Wang Laboratories
Warner Electric
WCP, Inc.
Wescal Industries, Inc., Data Systems Div.
Western Telematic Inc.
Western Union Telegraph Company
John Wiley & Sons, Inc.
Wilkinson Computer Sciences, Inc.
Worldwide Computer Services Inc.
Wyle Laboratories—Computer Products Div.
Electronic Enclosures Div.

Xerox Corporation
Xynetics, Inc.

# AUTHOR INDEX