varian software handbook

Volume 1

Specifications Subject to Change Without Notice

varian data machines/a varian subsidiary
printed in USA © 1973

98 A 9952 201

June 1973

Contents

Introduction

TABLE OF CONTENTS

Introduction

INTRODUCTION

Varian offers the computer user a wide range of systems configurations, processors, and peripherals to perform a great variety of tasks.

For full use of this extensive hardware capability Varian also offers a choice of efficient field-proven software packages for simplified programming and operations. These Varian software packages provide language processors, operating systems and utility programs. With these software packages a user can concentrate on his own particular applications rather than managing the system's resources.

Language Processors

Languages developed for programming applications include those processed by assemblers and compilers. The usual distinction between an assembler and a compiler is generally made upon the closeness of the relation of the source-language statements to the code they generate. An assembly language is closer to the executable code.

Assemblers

An assembler produces executable machine-language binary code from a symbolic form of statements, but the ratio of assembly-language statements to machine is frequently one to one instructions. This allows programming many of the machine's basic activities. A predominant reason for using the assembly languages is to control the system at that fundamental level.

The Varian 73/620 system assemblers (called DAS for " Data Assembler System") are available in three varieties. DAS 4A is designed for systems with only 4K words of central memory. DAS 8A is a more extensive and efficient assembler to run on system with 8K or more of main memory. DAS MR is a " macro" assembler which provides further extensions to the capablilties of the assembly language. All three handle the complete instruction set of the system but differ in the amount of programming convenience provided. For example, the DAS MR " macro" feature allows concise coding of elements within a program.

Compilers

In higher-level languages additional programming conveniences are available. In this category the Varian computer user has a choice of the widely-used versions of FORTRAN IV, BASIC or RPG IV. These languages are more removed from the machine instruction set than assembly languages. Translation of the higher-level languages is done by a compiler. The compilers produce many machine instructions for one statement so the programmers have an extremely concise mode of expression for their problem solutions.

One simple, easy-to-learn programming language is BASIC. With only a few hours of instruction, a person can program a Varian 73 or 620 computer and solve some simple problems. With continued use and greater understanding, the BASIC language can be applied to solve relatively sophisticated problems using its matrix operations.

FORTRAN IV is a widely-used problem-oriented language especially useful for scientific and mathematical applications. Varian's FORTRAN IV is compatible with the American National Standard Institute (ANSI) FORTRAN. Many routines and programs have been developed in FORTRAN IV and will save the user duplicating the time and effort.

For business applications the Report Program Generator (RPG) IV programming language provides several concise methods for handling alphameric data, and convenient means to do accounting and inventory programs.

All of Varian's higher-level languages provide interfaces with assembly-language routines. Through use of a combination of assembly language and a higher-level language the user retains the conveniences of the higher-level languages while gaining the particular control of the system at the level available only in assembly languages. (For example, a FORTRAN program could give a concise framework to DAS MR routines doing bit-level manipulation not available in FORTRAN).

Operating Systems

Two comprehensive software operating systems, MOS and VORTEX, are available for use with VARIAN 73 and 620 computers. Both systems incorporate a full repertoire of utility programs, as well as DAS MR, FORTRAN IV, and RPG IV.

MOS (Master Operating System) is specifically designed for batch-processing applications. The system provides input and output interfaces, operator communication, debugging aids, file maintenance and editing programs, and extensive reporting of systems errors and status.

VORTEX (Varian Omnitask Real-Time Executive) is a multi-programming system with special features for real-time applications. A number of different tasks may be stored in the main memory or on rotating-memory devices such as disc or drum. The tasks are scheduled by a resident executive program, which gives high priority to real-time

" foreground" tasks and lower priority to " background" tasks to be executed when there is time available. The scheduler uses the idle-time intervals embedded in most real-time applications to optimize use of the processor.

VORTEX increases the efficiency of any installation in which a computer is required to run a number of different programs in sequence. The user establishes the priority of the jobs to be executed, and then VORTEX automatically schedules and runs the programs without further operator intervention.

Organization of this handbook

This handbook is a compendium of manuals previously published as free-standing documents and provides additional information about the assembler.

Related Documentation

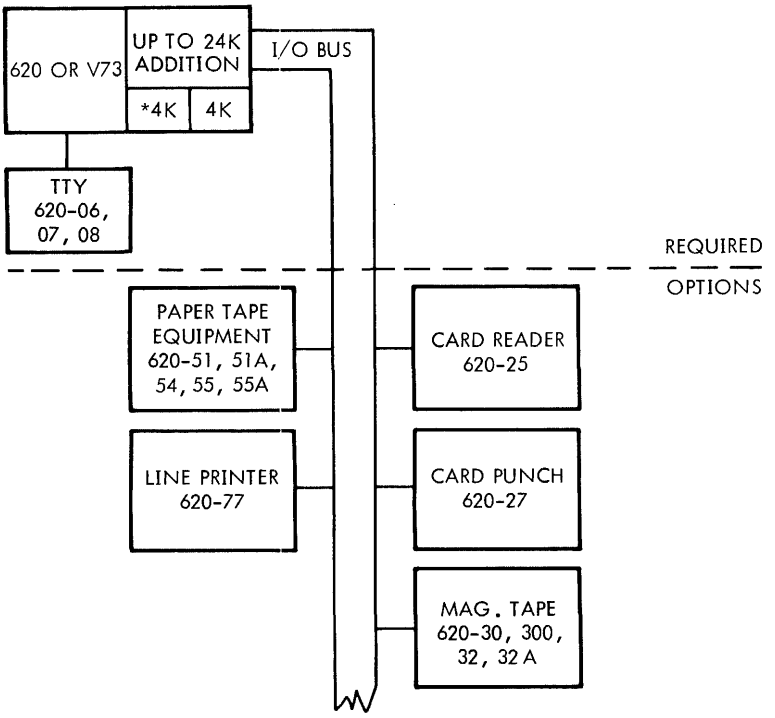
Varian's system handbooks provide a definition of the machines' instruction set and useful system information. The various handbooks and the document numbers are:

Handbook	Document number
Varian 73 System Handbook	98 A 9906 010
Varian 620-100 Computer Handbook	98 A 9905 003
Varian 620/f Computer Handbook	98 A 9908 002
Varian 620/L Computer Handbook	98 A 9905 000

Additional and more specific manuals can be located with the Publication Stock Number Catalog (98 A 9949 005) which lists document numbers of all publications.

System Configurations

The following diagrams indicate the types of hardware that can be used with the various software systems described in this volume.



*4K MEMORY REQUIRED FOR DAS 4A, 8K MEMORY REQUIRED FOR DAS 8A, DASMR WILL RUN IN 8K MEMORY WITH LIMITED CAPABILITIES. 12K IS RECOMMENDED FOR MOST APPLICATIONS.

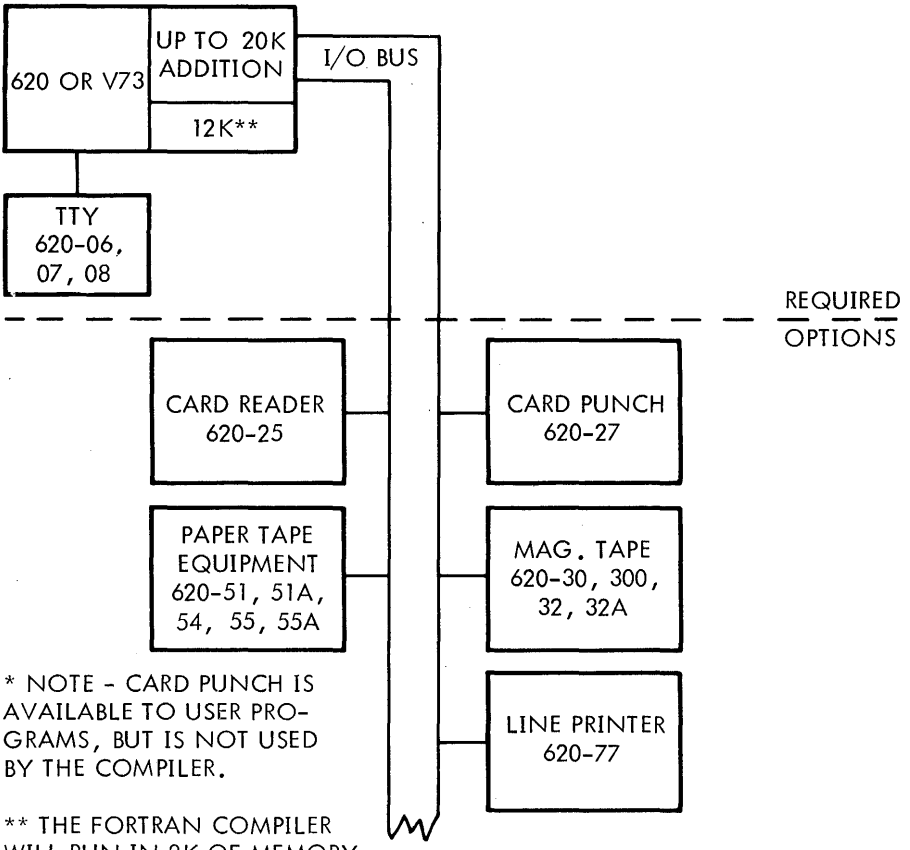
DAS-4A – This is a basic absolute assembler which will run in any VDM 620-series computer with 4K or more of memory. It includes a set of I/O drivers that are selected at load time. Therefore, one version will run with many different combinations of peripherals, as shown above.

DAS-8A – This assembler requires 8K or more of memory to run, and offers the user greater control over the assembly processes. Like DAS-4A, it includes a set of I/O drivers that are selected at load time.

DASMR – This is a free-standing version of the macro assembler used in MOS and VORTEX systems. It produces object code compatible with MOS.

VTII-1892

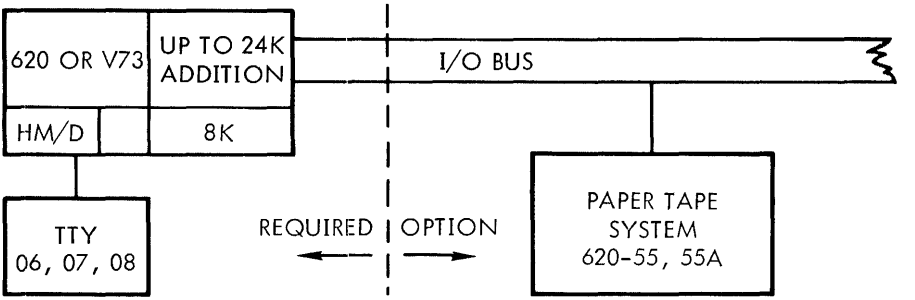
DAS 4A, 8A, DAS MR Assemblers (Stand-alone)



FORTRAN IV — This is an integrated software package consisting of a single-pass compiler, a relocating loader, and a set of runtime math and I/O routines. The compiler is fully compatible with ANSI Standard Fortran, and produces object code which is compatible with MOS.

VTII-1893

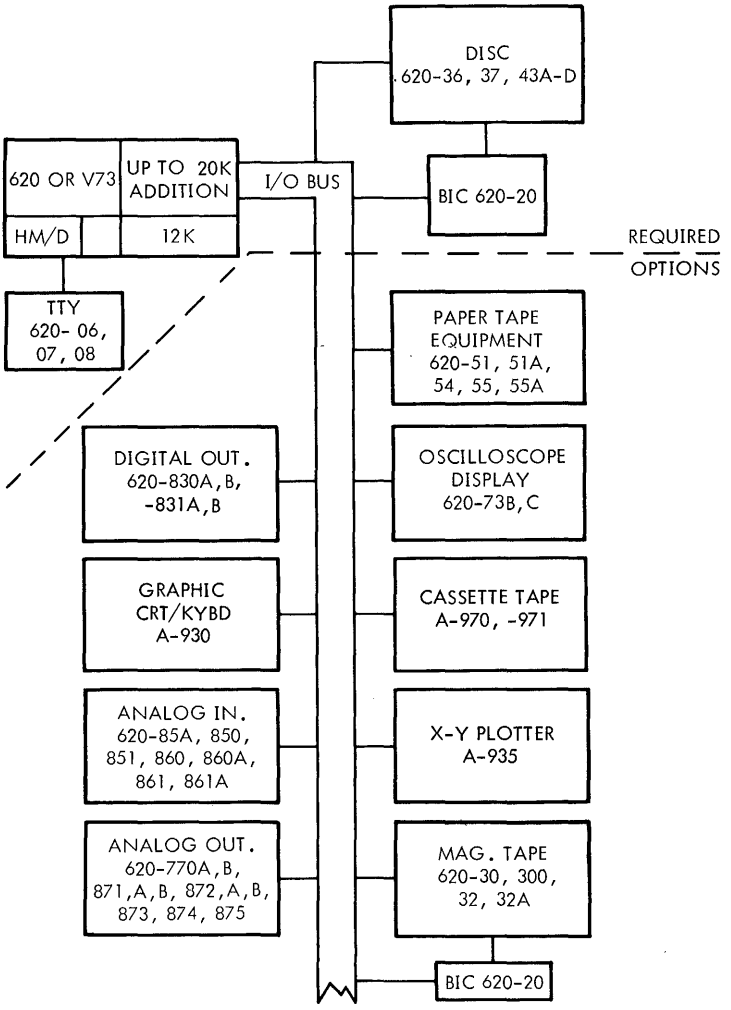
FORTRAN IV Compiler (Stand-alone)



BASIC – This version of the popular Dartmouth self-teaching language will run in any VDM 620-series computer with the hardware shown. It is applicable to a variety of business and scientific applications.

VTII-1894

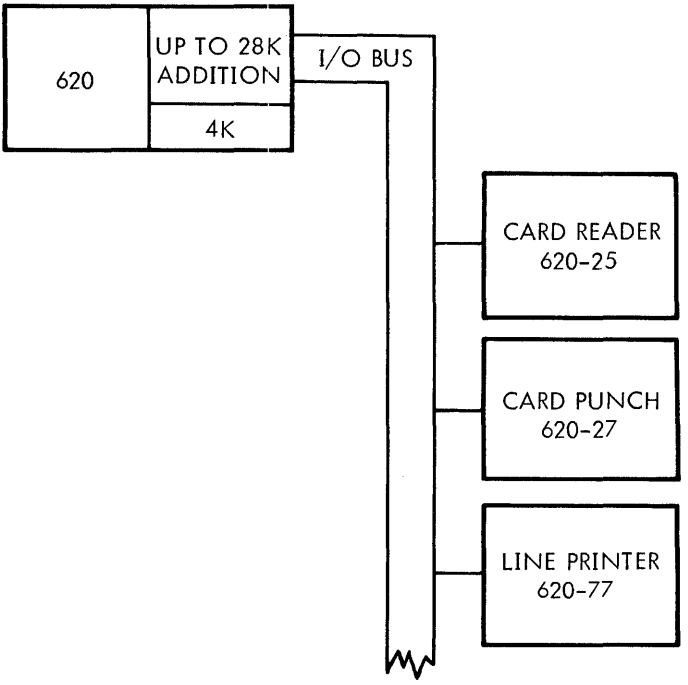
BASIC (Stand-alone)



EXTENDED BASIC expands on the BASIC language with special commands to control an external data acquisition and process control system, as shown in the diagram. In addition, directives have been included to allow the creation and control of files stored on a rotating memory device, and to facilitate chaining of program overlay segments.

VTII-1895

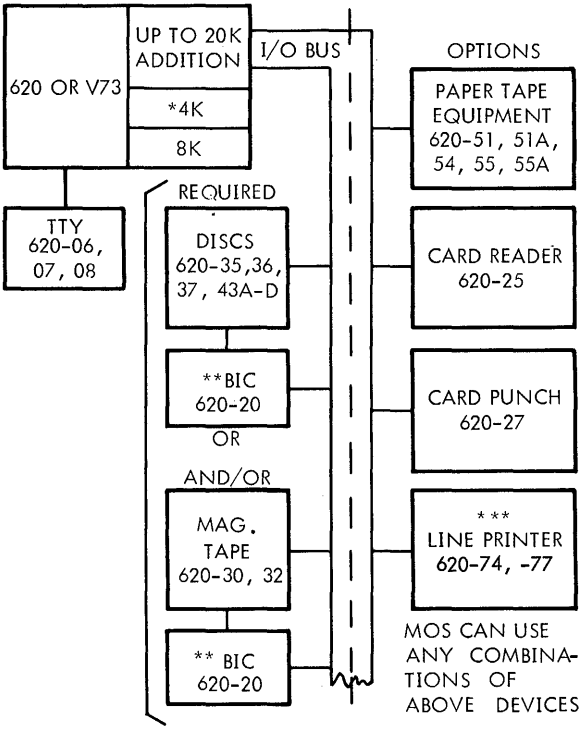
Extended BASIC



RPG-IV — This is an integrated software package consisting of a compiler, loader, and a set of runtime routines which provides a business language capability to the mini-computer user. RPG-IV is available both as a unit-record-oriented free-standing system and as a language processor under MOS.

VTII-1896

RPG-IV System



NOTE - ONE BIC MAY HANDLE UP TO 10 DEVICES, BUT FOR BEST SYSTEM PERFORMANCE, HIGH-TRANSFER RATE DEVICES SUCH AS DISCS SHOULD HAVE THEIR OWN BICs.

620-35 DISC REQUIRES BTC (E-2026H) INSTEAD OF BIC, AND MAY BE USED ON PMA CHANNELS OF V73, 620/f, AND 620/f-100 ONLY.

*8K REQUIRED FOR DASMR
12K REQUIRED FOR FORTRAN IV, RPG IV
16K REQUIRED FOR PERT

**BIC OPTIONAL

***620-74 SUPPORTED AS LINE PRINTER ONLY

MOS — MOS is a disc-, drum-, or magnetic tape-based batch operating system, which can be used with any VDM 620-series computer. It supports FORTRAN IV, DASMR, and RPG-IV; and it provides the user with RMD file management as well as automatic scheduling from the job stream.

VTII-1897

M.O.S.

DAS Assemblers

TABLE OF CONTENTS

DAS ASSEMBLERS

DAS ASSEMBLERS

The **Varian 73/620 assembler language (DAS)** translates symbolically coded instructions, directives, and data (source program) into their binary machine-language equivalents (object program). DAS allows the programmer to specify instructions, addresses, address modifications, and constants in a manner that is straightforward and meaningful to the computer.

Using DAS, the programmer generates a source program by coding instruction and directive mnemonics rather than numerical values. Memory addresses can be referenced symbolically, thus providing flexibility not attainable with absolute addressing. Constants can be used without prior conversion to binary or octal values. For ease in checkout and program documentation, comments can be added between symbolic source statements, or appended to the statements themselves.

DAS coding reduces machine-language bookkeeping to fully utilize computer capabilities without a corresponding compromise of an increase in the time required for programming.

Three versions of DAS are available:

a. DAS 4A operates in a minimum-configuration Varian 73 system comprising the computer, 4K of memory, and an on-line Teletype.

b. DAS 8A requires a minimum of 8K of memory and has extended capabilities compared to DAS 4A. Both DAS 4A and DAS 8A can operate with additional system peripherals.

c. DAS MR is a macro assembler, which produces relocatable object code, that can be loaded into any area of memory. DAS MR is available either as a free-standing program or as an integral part of the MOS or VORTEX operating system.

DAS processes source programs in two passes. The first pass defines user-designated symbols. The second pass produces an assembly listing and the object program.

Character Set

The DAS character set comprises:

Alphabetical characters

ABCDEFGHIJKLMNOPQ
RSTUVWXYZ

Numerical characters

0123456789

Teletype characters

CR (Carriage return)
LF (line feed)

Special characters

+	(plus sign)
–	(minus sign)
*	(asterisk)
/	(slash)
.	(period)
	(blank)
@	(at sign)
[	(left bracket)
]	(right bracket)
<	(less than)
>	(greater than)
↑	(up arrow)
←	(left arrow)
=	(equal sign)
,	(comma)
'	(prime)
(	(left parenthesis)
)	(right parenthesis)
\	(backslash)
!	(exclamation point)
"	(quotation mark)
#	(pound sign)
%	(percent sign)
&	(ampersand)
:	(colon)
;	(semicolon)
?	(question mark)
$	(dollar sign)

Format

DAS source programs are sequences of source statements (records). Each source statement comprises a combination of label, operation, variable, and comment fields, depending on the requirements of the computer instruction or assembler directive, and except in certain cases (described later in this section) generates one computer word.

Label Field

Symbols in the label field identify program points for reference by other parts of the program. They make a program point or particular numeric value more easily identifiable. The first appearance of a symbol in the label field establishes its identity throughout the remainder of the program. A previously established symbol is referenced by placing it in the variable field of the source statement, where DAS substitutes the previously assigned value from its symbol table.

For DAS 4A and DAS 8A, symbols in the label field comprise one to four alphanumeric characters for DAS MR there are from one to six such characters. The first character of a symbol is an alphabetic character, pound sign (#), or dollar sign. The following characters, if any, are chosen from the alphabetic, numeric subset, pound sign, and dollar sign. (The dollar sign and pound sign are used in the Varian software and should not be used in normal users programs). While only the given number of characters are recognized by DAS, additional characters can be added for programming convenience and/or documentation.

Symbols are usually attached only to those source statements referenced elsewhere in the program, but this is not mandatory.

Operation Field

This source statement field contains mnemonics for computer instructions (section 16) and assembler directives (defined later in this section). An asterisk following the mnemonic specifies **indirect addressing** (section 15). The mnemonics can be redefined with OPSY assembler directives (see below).

Variable Field

The purpose of this field varies according to the requirements of the operation defined by the source statement. The variable field can contain a symbol, a constant, or an expression combining symbols and constants.

DAS Expressions are similar to arithmetic expressions except that parentheses are not used. The variable field can contain the following operators.

+	(addition)
-	(subtraction)
*	(multiplication)
/	(division)

Arithmetic operations always involve all 16 bits of the computer words, and are performed from left to right, with multiplication and division occurring before addition and subtraction. Thus, A + B/C * D in DAS is equivalent to A + (B/C) *D in conventional notation.

Coding an asterisk in the first position of the variable field gives access to the then current value of the program location counter. Such an asterisk immediately precedes another operator, and this is the only case in which two adjacent operators are permitted in DAS. The asterisk is translated as the current program location (i.e., * + 1 means the current program location plus one).

In the following descriptions of DAS constants, *unsigned numbers are considered positive* DAS recognizes decimal and octal integers; floating-point numbers; alpha, address, and indirect address constants; and literals.

A **decimal integer** is a signed or unsigned string of from one to five decimal digits. the first of which cannot be zero (so as not to be confused with octal integers).

Example:

1 29 -3 -9000

An **octal integer** is a signed or unsigned string of from one to seven octal digits, the first of which is zero.

Example:

07 -044 +022745

A **floating-point number** has the form:) ± integer.fraction ± exponent, where the right parenthesis, at least one digit, and the decimal point are always present. Other items in the format are optional.

Examples:

)0375.64E+7)9.E-2,).1E+12
)-4.+20

Floating point numbers are not available under DAS 4A.

An **alpha constant** is a string of characters within primes ('), where, within DAS each character is represented in eight-bit ASCII code. Thus, each 16-bit memory address can hold two characters. Note that blanks are also recognized as characters.

In DAS 4A and DAS 8A, an alpha constant can be a term in an arithmetic expression. However, if more than one word is generated by the constant, only the last word is subject to arithmetic manipulation.

Examples:

'A'*0400 'AB'+1 'ABCD'+011

where, in the last example, two words are generated and 011 added to the second word.

An **address constant** is a symbol, number, or expression enclosed in parentheses. It generates a 15-bit direct address (bit 15 = 0).

Examples:

(aaaa+2) (31) (aaaa)

where aaaa is an address symbol whose value is taken from the symbol table by DAS.

An **indirect address constant** is an *address contant* followed by an asterisk. It generates a 15-bit indirect address (bit 15 = 1).

Examples:

(aaaa+2)* (3)* (aaaa)*

Literals provide a method for creating and referencing data by expressing the value of the information instead of its address. DAS determines the address and inserts it in the referencing statement and generates a literal table, discarding duplicate values in the table.

A literal is any format of a one-word constant preceded by an equal sign. In a statement requiring more than one literal, they are separated by commas.

Examples:

=29 =-044 =(aaaa+2)*
='GO' ='A'

Comments Field

This field is used for programming notes. An entire source statement can be commentary if an asterisk is coded in the first position. The assembler ignores all comments in the assembly process, but lists them with the program listing output.

Computer Instructions

DAS assemblers recognize the complete instruction sets of all Varian 73/620 computers, even when the system on which they operate lacks the hardware for executing a particular instruction. **The programmer, therefore, must have a thorough knowledge of the instructions applicable to his system before attempting to assemble a program.**

Computer instructions are described in detail in the system handbook for each particular system.

In this section, all Varian 73/620 instructions are divided into five types, according to assembler format requirements.

All Varian 73/620 instructions in DAS have the general field format

Label Operation Variable Comments

where the label field is optional and contains a symbol when used; the operation field contains the instruction mnemonic; the variable field contains one, two, or three expressions (separated by commas when there is more than one), and the comments field is optional.

Addressing

If an assembler source statement specifies an address in the first 2,048 words of memory without indirect addressing, the assembler generates an instruction with direct addressing.

If indexing is specified, the assembler generates an indexed instruction.

Specifying indirect addressing with a data address lower than 512 generates an instruction with indirect addressing and the specified effective memory address.

In all other cases, including indirect addressing with an address higher than 511, the assembler generates an instruction with indirect addressing and the specified effective memory address, stores the address in a table, and inserts the storage address in the referencing instruction. Duplicate values in the table are discarded.

In the Varian 73/620, indirect addressing is limited to five levels with one-word instructions and to four levels with two-word instructions.

Instruction Types

Table 1 summarizes the characteristics of the five types of computer instructions for DAS use. Instruction mnemonics are given in the applicable type description below and summarized in table 2.

Table 1. Assembler Instruction Type Characteristics

Parameter	Type 1	Type 2	Type 3	Type 4	Type 5
Words generated	1	2	2	1	2
Memory addressed	Yes	Yes*	Yes	No	Yes
Indirect addressing	Yes	Yes*	Yes	No	Yes
Indexing	Yes	No	No	No	Yes
Variable field expressions	1 or 2	1	2	1	1 to 3
Microcoding	No	No	Yes	Yes	No

* Except for immediate instructions.

Table 2. Summary of Assembler Instruction Types

Type 1	Type 2		Type 3	Type 4		Type 5
ADD	ADDI	JS3NM	BT	AOFA	LASR	ADDE
ANA	ANAI	JXZ	IME	AOFB	LLRL	ANAE
DIV	DIVI	JXZM	JIF	AOFX	LLSR	DIVE
ERA	ERAI	LDAI	JIFM	ASLA	LRLA	ERAE
INR	INRI	LDBI	JMIF	ASLB	LRLB	IJMP
LDA	JAN	LDXI	OME	ASRA	LSRA	INRE
LDB	JANM	MULI	SEN	ASRB	LSRB	JSR
LDX	JANZ	ORAI	XIF	CIA	MERG	LDAE
MUL	JANZM	STAI		CIAB	NOP	LDBE
ORA	JAP	STBI		CIB	OAB	LDXE
STA	JAPM	STXI		COMP	OAR	MULE
STB	JAZ	SUBI		CPA	OBR	ORAE
STX	JAZM	XAN		CPB	ROF	SRE
SUB	JBZ	XANZ		CPX	SEL	STAE
	JBZM	XAP		DAR	SEL2	STBE
	JMP	XAZ		DBR	SOF	STXE
	JMPM	XBNZ		DECR	SOFA	SUBE
	JOF	XBZ		DXR	SOFB	
	JOFM	XEC		EXC	SOFX	
	JOFN	XOF		EXC2	TAB	
	JOFNM	XOFN		HLT	TAX	
	JSS1	XS1		IAR	TBA	
	JSS2	XS1N		IBR	TBX	
	JSS3	XS2		INA	TXA	
	JS1M	XS2N		INAB	TXB	
	JS1NM	XS3		INB	TZA	
	JS2M	XS3N		INCR	TZB	
	JS2NM	XXNZ		LASL	ZERO	
	JS3M	XXZ				

Assembler type 1 instructions are:

ADD	LDA	STA
ANA	LDB	STB
DIV	LDX	STX
ERA	MUL	SUB
INR	ORA	

An assembler type 1 instruction occupies one computer word and is memory-addressing.

Indirect addressing is specified by an asterisk after the mnemonic or after a variable field expressed in parentheses.

Examples:

```
LDA*  expression
LDA   (expression)*
```

Indexing is specified by two expressions in the variable field. The first is the indexing increment and is less than 0512. The second specifies the indexing register: X register = 1, and B register = 2. These instructions cannot be postindexed.

Example:

```
LDA   0300,1
```

loads the A register with the contents of the memory address specified by the sum of the X register contents and 0300. Thus, if the X register contains 0200, the operand for this instruction is in memory address 0500.

Assembler type 2 instructions are:

ADDI	JOFN	STXI
ANAI	JOFNM	SUBI
DIVI	JSS1	XAN
ERAI	JSS2	XANZ
INRI	JSS3	XAP
JAN	JS1M	XAZ
JANM	JS1NM	XBNZ
JANZ	JS2M	XBZ
JANZM	JS2NM	XEC
JAP	JS3M	XOF
JAPM	JS3NM	XOFN
JAZ	JXZ	XS1
JAZM	JXZM	XS1N
JBZ	LDAI	XS2
JBZM	LDBI	XS2N
JMP	LDXI	XS3
JMPM	MULI	XS3N
JOF	ORAI	XXNZ
JOFM	STAI	XXZ
	STBI	

An assembler type 2 instruction occupies two consecutive computer words and is memory-addressing. The second word is the address of a jump, jump-and-mark, or execution instruction or the operand specified by an immediate instruction.

Indirect addressing is specified as with an assembler type 1 instruction. These instructions cannot be indexed.

Assembler type 3 instructions are:

BT	JIFM	SEN
IME	JMIF	XIF
JIF	OME	

An assembler type 3 instruction occupies two consecutive computer words and is memory-addressing. It differs from an assembler type 2 instruction in that the variable field contains two expressions to implement instruction microcoding as described below.

For the JIF, JIFM, JMIF, and XIF instructions, the first expression specifies the conditions required for the jump, jump-and-mark, or execution. The conditions(s) are specified according to the rules given in section 16 and summarized below. As indicated, multiple conditions can be specified by setting additional bits.

Variable Field	Jump/Execute if:
0001	Overflow indicator is set
0002	A register contents are positive
0004	A register contents are negative
0010	A register contents are zero
0020	B register contents are zero
0040	X register contents are zero
0100	SENSE switch 1 is set
0200	SENSE switch 2 is set
0400	SENSE switch 3 is set

Example:

```
JIF   0222,ALFA
```

Takes the next instruction from symbolic address ALFA if the A register contains a positive number (0002), the B register contains zero (0020), and SENSE switch is set (0200); i.e., 0002 + 0020 + 0200 = 0222.

For the SEN instruction, the first expression specifies the device address and the I/O function; for IME and OME, the device address.

For the BT instruction, the first expression specifies the register and bit to be tested.

Example:

```
BT    056,ADDR
```

takes the next instruction from symbolic address ADDR if bit 14 of the A register contents is zero.

Indirect addressing is specified by an asterisk after the mnemonic or after a variable field expression in parentheses as described for the type 1 instructions. **Note:** IME and OME cannot specify indirect addressing.

Assembler type 4 instructions are:

AOFA	EXC2	OAR
AOFB	HLT	OBR
AOFX	IAR	ROF
ASLA	IBR	SEL
ASLB	INA	SEL2
ASRA	INAB	SOF
ASRB	INB	SOFA
CIA	INCR	SOFB
CIAB	IXR	SOFX
CIB	LASL	TAB
COMP	LASR	TAX
CPA	LLRL	TBA
CPB	LLSR	TBX
CPX	LRLA	TXA
DAR	LRLB	TXB
DBR	LSRA	TZA
DECR	LSRB	TZB
DXR	MERG	TZX
EXC	NOP	ZERO
	OAB	

An assembler type 4 instruction occupies one computer word and does not address memory.

For COMP, DECR, INCR, MERG, and ZERO and the register transfer/modification instructions, the assembler generates an instruction as specified by the value in the

variable field. This value is determined by coding the summed octal value of the possible binary configurations described for these instructions in the systems handbook.

Example:

```
COMP    035
```

unconditionally takes the inclusive-OR and complements the contents of the A (0010) and B (0020) registers, and places the result in the A (0001) and X (0004) registers. Note that if bit 8 were one in the above example the instruction is executed only if the overflow indicator is set.

For EXC, SEL, EXC2, and SEL2, the expression specifies the I/O function and the device address; for the remainder of the I/O instructions in this group, the device address only (the I/O function being specified by the mnemonic).

Example:

```
CIB     030
```

clears the B register and loads it from peripheral specified by the device address 030 (standard device addresses are given in the systems handbooks).

Note: SEL/SEL2 are identical to EXC/EXC2 instructions.

Assembler type 5 instructions are:

ADDE	INRE	SRE
ANAE	LDAE	STAE
DIVE	LDBE	STBE
IJMP	LDXE	STXE
ERAE	MULE	SUBE
JSR	ORAE	

An assembler type 5 instruction occupies two consecutive computer words and is memory-addressing.

Indirect addressing is specified by an asterisk after the mnemonic or after a variable field expression in parentheses as described for the type 1 instructions.

Preindexing the V73 and 620 instructions is specified as described for the type 1 instructions. Note that IJMP and SRE cannot be preindexed.

Postindexing the V73 and 620 instructions is specified by three expressions in the variable field. The first expression is the data address, the second specifies the indexing register (X register = 1, and B register = 2), and the third is logically ORed with the instruction

word to set bit 7 (which specifies postindexing). The assembler does not check the validity of the third expression, thus one should always use the value 0200.

Example:

```
LDAE        ADDR,2,0200
```

loads the A register extended and postindexed with the B register.

JSR can be neither preindexed nor postindexed.

For SRE, the first expression in the variable field is the data address, the second specifies the type of addressing (1 = indexed with X, 2 = indexed with B, and 7 = direct /indirect), and the third is logically ORed with the instruction word to control bits 3-5 to specify the register to be compared (010 = A register, 020 = B register, and 040 = X register). Note that indirect addressing is specified by an asterisk following the instruction mnemonic.

Examples:

```
SRE    ADDR,7,020
```

compares the contents of the B register with the directly addressed word at ADDR, and, if equal, skips the next two locations.

```
SRE*   ADDR,1,010
```

compares the contents of the A register with the word at ADDR, using indirect addressing and postindexing with the X register.

Assembler Directives

Directives are instructions to the assembler. They are divided into the following functional groups:

- Symbol definition
- Instruction definition
- Location counter control
- Data definition
- Memory reservation
- Conditional assembly
- Assembler control
- Subroutine control
- List and punch control
- DAS 8A interface to stand-alone FORTRAN
- Program linkage
- MOS I/O control
- Macro definition

Assembler directives have the same general format as the computer instructions. In the following descriptions of the individual directives, the field format

label operation variable

is used, with the optional comment field being understood to follow the variable field when used. In cases where the variable field contains more than one item or expression, these are always separated by commas. Mandatory elements of the directive are in **bold type**, and optional items, in *italic type*.

Table 3 summarizes the assembler directives (arranged by function) and indicates those recognized by each DAS assembler.

Table 3. Directives Recognized by DAS Assemblers

Function	Directive	DAS 4A	DAS 8A	DAS MR
Symbol definition	EQU	Yes	Yes	Yes
	SET	Yes	Yes	Yes
	MAX	No	Yes	No
	MIN	No	Yes	No
Instruction definition	OPSY	No	Yes	Yes
Location counter control	ORG	Yes	Yes	Yes
	LOC	Yes	Yes	Yes
	BEGI	Yes	Yes	Yes
	USE	No	Yes	No
Data definition	DATA	Yes	Yes	Yes
	PZE	Yes	Yes	Yes
	MZE	Yes	Yes	Yes
	FORM	No	Yes	Yes
Memory reservation	BSS	Yes	Yes	Yes
	BES	Yes	Yes	Yes
	DUP	No	Yes	Yes
Conditional assembly	IFT	No	Yes	Yes
	IFF	No	Yes	Yes
	GOTO	No	Yes	Yes
	CONT	No	Yes	Yes
	NULL	No	Yes	Yes
Assembler control	MORE	Yes	Yes	No
	END	Yes	Yes	Yes

Table 3. Directives Recognized by DAS Assemblers *(continued)*

Function	Directive	DAS 4A	DAS 8A	DAS MR
Subroutine control	ENTR	Yes	Yes	Yes
	RETU*	Yes	Yes	Yes
	CALL	Yes	Yes	Yes
List and punch control	LIST	No	Yes	No
	NLIS	No	Yes	No
	SMRY	No	Yes	Yes
	DETL	No	Yes	Yes
	PUNC	No	Yes	No
	NPUN	No	Yes	No
	SPAC	No	Yes	No
	EJEC	No	Yes	Yes
	READ	No	Yes	No
Program linkage	NAME	No	No	Yes
	EXT	No	No	Yes
	COMM	No	No	Yes
MOS I/O control	See "MOS I/O Control" in MOS section			
Macro definition	MAC	No	No	Yes
	EMAC	No	No	Yes

Symbol Definition Directives

These directives assign arbitrary values to symbols in the symbol table. This table is a list of symbols appearing in the source program. For each symbol in the table, there is a corresponding value, usually an address in memory. symbol table capacities are summarized in table 4.

Table 4. DAS Symbol Table Capacities

Assembler	4K Memory	8K Memory	>8K Memory
DAS 4A	150	1,450	1,450 + n (1,300)
DAS 8A	-----	440	440 + n (800)
DAS MR	-----	20	20 + n (800)

where n = number of 4K memory increments above 8K.

EQU (DAS 4A, DAS 8A, DAS MR)

This directive has the format

symbol EQU expression

It places the **symbol** in the assembler's symbol table and assigns it the value of the **expression**. If the symbol has already been entered in the symbol table, DAS outputs error message ***DD** (described later in this section), and the expression replaces the value in the symbol table. If a symbol is used as the variable field expression, it must have been previously defined. **The label field symbol is mandatory.**

SET (DAS 4A, DAS 8A, DAS MR)

This directive has the format

symbol SET expression

It is the same as EQU, except that there is no error message output if the symbol has already been entered in the symbol table.

MAX (DAS 8A)

This directive has the format

symbol MAX expression, *expression(s)*

It assigns the largest algebraic value found among the **expressions** to the **symbol**. If a symbol is used as a variable field expression, it must have been previously defined. The label field symbol is mandatory. Use SET to redefine the symbol.

MIN (DAS 8A)

This directive has the format

symbol MIN expression,*expression(s)*

It is the same as MAX, except that the symbol is assigned the smallest algebraic value found among the **expressions**.

Instruction Definition Directive

This directive redefines a standard instruction mnemonic.

OPSY (DAS 8A, DAS MR)

This directive has the format

symbol OPSY mnemonic

It makes the symbol a mnemonic with the same definition as the variable field mnemonic.

Example:

```
CLA   OPSY LDA
      CLA  BETA
```

Location Counter Control Directives

These directives control the program location counter(s), which control memory area assignments and always point to the next available word.

DAS 8A has several location counters and directives to modify or preset their values. Table 5 lists the five standard DAS 8A location counter symbols and their uses. They need not be created by the user. However, up to eight other location counters can be created, thus providing complex relocatable and overlay programs within a single assembly. Relocatability rules are given later in this section.

There are no user-created location counters at the beginning of an assembly. The assembler uses three location counters for program location assignment. Thus, IAOR (indirect pointer assignments) and LTOR (literal assignments) are always in use, as is a third counter used to assign locations to generated instructions and data. The blank location counter performs this task until the USE directive specifies another counter.

In a straightforward program using only one location counter, the ORG and LOC directives completely control the counter.

ORG (DAS 4A, DAS 8A, DAS MR)

This directive has the format

symbol **ORG expression**

It sets the location counter currently in use to the value of the **expression.** If a symbol is present in the label field, it is also set to the value of the expression.

Any symbol used as the variable field expression must have been previously defined.

Table 5. Standard DAS 8A Location Counters

Counter	Initial Value	Description
COMN	002000	Controls assignment of memory within an interface area common to two or more programs
IAOR	000200	Controls assignment of memory to indirect pointers
LTOR	001000	Controls assignment of memory to literals
SYOR	000000	Controls assignment of memory to all system parameters
blank	004000 000000 for 4A	Used initally and normally by the as- unless overridden by the USE or ORG directive

LOC (DAS 4A, DAS 8A, DAS MR)

This directive has the format

symbol **LOC expression**

It is used if the data and instructions following this LOC address are to be moved to the LOC address by the object program before execution i.e., to keep a block of data or instructions undisturbed by assembly. Data or instructions following LOC are generated as if an ORG directive had changed the current location counter value. However, this value is not actually changed.

Any symbol used as a variable field expression must have been previously defined. LOC cannot be used in a relocatable program.

BEGI (DAS 4A, DAS 8A)

This directive has the format

symbol **BEGI expression**

It creates a new location counter, or redefines the value of any location counter before the counter has been used. BEGI gives the new or redefined location counter the value of the **expression**, but has no effect on the current location counter.

BEGI cannot redefine the value of any location counter that has been used for location assignment.

Any symbol used as a variable field expression must have been previously defined.

USE (DAS 8A)

This directive has the format

blank USE xxxx

where **xxxx** is a blank, COMN, SYOR, or a user-created location counter label.

The USE directive uses location counter **xxxx** to assign locations to data and instructions (except literals and indirect pointers).

If **xxxx** is PREV, the previously used location counter is recalled with the restriction that only the last-used counter can be so recalled.

Data Definition Directives

These directives control the sign and assignment of data words. In the descriptions, **item** refers to a data item, which can be an expression or a direct or indirect address.

DATA (DAS 4A, DAS 8A, DAS MR)

This directive has the format

symbol **DATA item**,*item(s)*

It generates data words with the values specified by the **items** in the variable field. DATA assigns the *symbol*, if used, to the memory address of the first generated word. In the absence of a symbol, an unlabeled block of data is generated.

When a single alpha constant is used in the variable, DAS 4A and DAS MR left-justify it in the field and fill the remaining positions with blanks, and DAS 8A right-justifies it, filling the remaining positions with zeros.

PZE (DAS 4A, DAS 8A, DAS MR)

This directive has the format

symbol **PZE item**,*item(s)*

It is similar to DATA except that the sign bit of the generated data word is always zero (positive).

MZE (DAS 4A, DAS 8A, DAS MR)

This directive has the format

symbol **MZE** **item,***item(s)*

It is similar to DATA except that the sign bit of the generated data word is always one (negative).

FORM (DAS 8A, DAS MR)

This directive has the format

symbol **FORM** **term,***term(s)*

where the **terms** are absolute terms or expressions.

FORM specifies the format of a bit configuration of a data word. The *symbol*, if used, is the name of the format. The **terms** specify the length in bits of each field in the generated data word, where the sum of their values is from one to the number of bits in the computer word.

FORM is ignored if there are any errors in the variable field, except that an error is flagged when a **term** cannot be represented in the number of bits specified when FORM is applied (by placing its name in the operation field of a symbolic source statement) to another statment. FORM can be redefined.

Example:

```
BYTE    FORM    8,8
BCD     FORM    4,4,4,4
PTAB    FORM    1,2,3,4
```

would, given the FORM definition

```
ABC     FORM    6,2,8
```

and the FORM reference

```
ABC     FORM    2*3,1,'A'
```

generate the binary data word

```
0   001   100   111   000   001
```

Memory Reservation Directives

These directives control the reservation of memory addresses and areas.

BSS (DAS 4A, DAS 8A, DAS MR)

This directive has the format

symbol **BSS** **expression**

It reserves a block of memory addresses by increasing the value of the current location counter the amount indicated by the **expression**. The *symbol*, if used, is assigned the value of the counter prior to such an increase, thus referencing the starting address of the reserved block.

The location counter always points to the next available word.

If the variable field expression value is zero, the *symbol* is assigned the next available address.

BES (DAS 4A, DAS 8A, DAS MR)

This directive has the format

symbol **BES** **expression**

It is similar to BSS, except that if there is a symbol it is assigned to the address one less than the incremented location counter. If the variable field expression is zero, the symbol is assigned the last available address.

DUP (DAS 8A, DAS MR)

This directive has the formats

blank **DUP** **n**
blank **DUP** **n,m**

It duplicates source statements following its use. The first format duplicates the next source statement the number of times specified by **n**. The second format duplicates the next source statement (the number of which is specified by **m**) the number of times specified by **n**, where **m**, ≤ 3 and **n** $\leq 32{,}767$. If **n** or **m** is zero, it is treated as if it were a one.

Conditional Assembly Directives

These directives assemble portions of the program according to the conditions specified in the variable fields.

IFT (DAS 8A, DAS MR)

This directive has the format

blank **IFT** **expression**,*expression(s)*

It assembles the next symbolic source statement only if the first **expression** is less than the second, and the second is less than or equal to the third.

Examples:

IFT a

for $a \neq 0$.

IFT a,,b

for $a \neq b$.

IFT a,b,b

for $a < b$.

IFT 0,a,b

for $0 < a \leq b$.

IFF (DAS 8A, DAS MR)

This directive has the format

blank **IFF** **expression**,*expression(s)*

It is similar to IFT (IFT = true), except that IFF (IFF = false) is the logical complement of IFT.

Examples:

IFF a

for $a = 0$.

IFF a,,b

for $a = b$.

IFF a,b,b

for $a \geq b$.

IFF 0,a,b

for $0 \geq a > b$.

GOTO (DAS 8A, DAS MR)

This directive has the formats

blank GOTO symbol
blank GOTO symbol,
blank GOTO integer
blank GOTO integer,

It skips more than one instruction and usually follows an IFF or IFT directive. All source statements between the GOTO and the statement containing the **symbol** in its lable field are skipped, and the instruction so labled executed next. GOTO cannot return to an earlier point in the program.

If the first and third GOTO formats are used, the skipped instructions are listed. If the second and fourth formats (containing a comma after the variable field element) are used, they are not listed. This listing can also be suppressed by a SMRY directive.

CONT (DAS 8A, DAS MR)

This directive has the format

symbol CONT blank

It provides a target for a previous GOTO directive. The **symbol** is not entered in the assembler's symbol table.

NULL (DAS 8A, DAS MR)

This directive has the format

symbol NULL blank

It provides a target for a previous GOTO directive with the **symbol** entered in the symbol table. NULL has the same effect as a BSS directive with a blank variable field.

Assembler Control Directives

These directives signal the end or continuance of an assembly.

MORE (DAS 4A, DAS 8A)

This directive has the format

blank MORE blank

It halts the assembly process to allow additional source statements to be put in the input device. Assembly resumes when the RUN or START switch on the computer control panel is pressed. MORE is never listed.

END (DAS 4A, DAS 8A, DAS MR)

This directive has the format

blank **END** *expression*

It is the last source statement in the program. The *expression* is the execution address of the program after it has been loaded into the computer. A blank in the variable field yields an execution address of 000000.

Subroutine Control Directives

These directives create closed subroutines and control their use.

ENTER (DAS 4A, DAS 8A, DAS MR)

This directive has the format

symbol ENTRblank

where the **symbol** is the name of the subroutine called. ENTR generates a linkage word of zero in the object program.

RETU* (DAS 4A, DAS 8A, DAS MR)

This directive has the format

symbol *RETU** *expression*

It returns from a closed subroutine, generating an unconditional jump to the address indicated by the value of the expression.

CALL (DAS 4A, DAS 8A, DAS MR)

This directive has the format

symbol **CALL** **name**,*parameter*,*error*

where

name	is the symbolic name of a subroutine
parameter	is an optional list of parameters comprising valid data items
error	is an optional list of error returns comprising valid data items

CALL causes the program to jump to the closed subroutine specified by **name**. Where a *symbol* is used in the label field, it is entered in the symbol table and assigned the value of the current location counter.

Example:

```
CALL    FUNC,X,Y+1,(ERR),(GOOF)*
```

produces a machine code identical to that obtained with

```
JMPM    FUNC
DATA    X,Y+1,(ERR),(GOOF)*
```

List and Punch Control Directives

These directives, which are operative only during the second pass of the assembler (that producing the object program and listings), control listing and punching during program assembly.

List (DAS 8A)

This directive has the format

blank **LIST** **blank**

It causes the assembler to produce a program listing. The assembler normally outputs a lost of the source statements. The LIST directive is used to bring the assembler back to this condition when any of the following directives change the listing status.

NLIS (DAS 8A)

This directive has the format

blank NLIS blank

It suppresses further listing of the program.

SMRY (DAS 8A, DAS MR)

This directive has the format

blank SMRY blank

It suppresses the listing of source statements that have been skipped under control of the conditional assembly directives.

DETL (DAS 8A, DAS MR)

This directive has the format

blank DETL blank

It removes the effect of SMRY, i.e., causes listing of all source statements, including those skipped by conditional assembly directives.

PUNC (DAS 8A)

This directive has the format

blank PUNC blank

It causes the assembler to produce a paper tape punched with the object program. The assembler normally outputs such a tape. PUNC returns the assembler to this condition when the following directive changes the punching status:

NPUN (DAS 8A)

This directive has the format

blank NPUN blank

It suppresses further production of paper tape punched with the object program.

SPAC (DAS 8A, DAS MR)

This directive has the format

blank SPAC blank

It causes the listing device to skip a line. SPAC is not listed.

EJEC (DAS 8A, DAS MR)

This directive has the format

blank EJEC blank

It causes the listing device to move to the next top of form. EJEC is not listed.

READ (DAS 8A)

This directive has the format

bland READ *number*

where

number	is the number of characters (20 to 80) from each source statement to be processed by the assembler

Normally, the assembler processes 80 characters per statement with 026 keypunch codes. If *number* is outside the range 20 to 80, the assembler resets the number of characters to 80 and outputs error message ***SZ**.

Unless there is an ***SZ** error message, the SMRY directive suppresses the listing of READ during the second pass of the assembly process.

Program Linkage Directives

These directives establish and control links among programs that have been assembled separately but are to be loaded and executed together.

This directive has the format

blank NAME symbol,...,*symbol*

It establishes linkage definition points among separately assembled programs. Each **symbol(s)** can then be referenced by other programs. Each **symbol** also appears in the label field of a symbolic source statement in the body of the program. Undefined NAME symbols cause error messages to be output.

Examples:

```
NAME    A
NAME    A,B
NAME    EX,WHY,ZEE
```

This directive has the format

symbol **EXT** *symbol,...,symbol*

In linking separately assembled programs, it declares each *symbol* not defined within the current program. Each *symbol*, in both the label and variable field, is output to the relocatable loader with the address of the last reference to the symbol.

If a symbol is not defined within the current program and not declared in an EXT directive, it is considered undefined and causes an error message output. If a symbol is declared in EXT but not referenced within the current program, it is output to the loader for loading, but no linkage to this program is established. If a symbol is both defined in the program and declared to be external, the EXT declaration is ignored.

Examples:

```
        EXT     AY
BEG     EXT     BE,SEE
        EXT     DEE,EE,FF,GEE
```

This directive has the format

symbol **COMN item**

where **item** is an absolute item or expression.

COMN defines an area in blank common for use at execution time. This allows an assembler program to reference the same blank common area as a FORTRAN program. The common area is cumulative for each use of COMN, i.e, the first COMN defines the base area of the blank common, the second COMN defines an area to be added to the already established base, etc.

Examples:

```
AAA     COMN    3
        COMN    6*2
BBB     COMN    9
```

MOS I/O Control Directives

As a free-standing program or under MOS, DAS MR accepts the MOS control directives listed below and explained in the Master Operating System section of this handbook.

Directive	Description
RBIN	Read binary record
RALF	Read alphanumeric record
RBCD	Read binary-coded decimal (BCD) record
WBIN	Write binary record
WALF	Write alphanumeric record
WBCD	Write BCD record
WEOF	Write end of file
REW	Rewind
SKFF	Skip files forward
SKFR	Skip files reverse
SKRF	Skip records forward
SKRR	Skip records reverse
FUNC	Function
STAT	Status
ION	I/O driver reference number

VORTEX I/O Control Directives

DAS MR accepts the VORTEX control directives that are listed below and explained in the VORTEX Reference Manual (document number 98 A 9952 101).

Directive	Description
OPEN	Open file
CLOSE	Close file
READ	Read one record
WRITE	Write one record
REW	Rewind
WEOF	Write end of file
SREC	Skip one record
FUNC	Function
STAT	Status
DCB	Generate data control block
FCB	Generate file control block

Macro Definition Directives

These directives begin and end macro definitions. The macro is the assembly equivalent of the execution subroutine. It is defined once and can then be called from the program. The macro is an algorithmic statement of a process that can vary according to the arguments supplied. It is assembled with the resultant data inserted into the program at each point of reference, whereas the subroutine executed during execution time appears but once in a program. Its definition comprises the statements between MAC and EMAC.

MAC (DAS MR)

This directive has the format

symbol MAC blank

It introduces a macro definition. The **symbol** is the name of the macro.

EMAC (DAS MR)

This directive has the format

blank EMAC blank

It terminates the definition of a macro.

A macro is called by the appearance of its name in the operation field of a symbolic source statement. The variable field of this statement contains expression(s) P(1), P(2),...P(n), then processed with the values in the table being substituted for the respective values of the expressions in the source statement variable field. For example, if the variable field of the symbolic source statement contains

2,B,9+8,=63

then within the generated macro P(1) = 2, P(2) is the value of B, P(3) = 021, and P(4) is the address of the value 63. All terms and expressions within the macro-referencing symbolic source statement parameter list are evaluated prior to calling the macro.

If the label field of such a source statement contains a symbol, the symbol is assigned the value and relocatability of the location counter at the time the macro is called but before data generation.

A macro definition can contain references to machine instruction mnemonics or to assembler directives other than DUP. Macros can be nested within macros to a depth limited only by the available memory at assembly time.

Example: Define the macro.

```
SBR   MAC
      SEN   0200 P(1),*+3
      JMP   *-2
      EMAC
```

Call the macro.

```
      SBR   031
```

Expand the macro.

```
      SEN   0231,*+3
      JMP   *-2
```

P(0) can also be accessed by a normal call. P(0) is the first entry in the table formed by the assembler and contains the number of entries in that table. The following example shows the output listing obtained by calling P(0):

```
                                1     A     MAC
                                2           DATA      P(0)
                                3           EMAC
000001      000000A             4           A
000002      000001A             5           A         1
000003      000002A             6           A         1,2
000004      000003A             7           A         1,2,3
000005      000004A             8           A         1,2,3,4
000006      000005A             9           A         1,2,3,4,5
                                10          END
```

Symbol And Expression Modes

Each symbol or expression has one of the following modes assigned by the assembler:

a. External (E)

b. Common (C)

c. Relative (R)

d. Absolute (A)

The mode of an expression is determined by the mode of the symbols in the expression.

The mode of a symbol is determined by the following rules:

a. If the symbol is in an EXT directive, the mode is E.

b. If the symbol is defined by a COMN directive, the mode is C.

(continued)

c. If the symbol is a symbol in a program, or if * is the current location counter value, the mode is R.

d. If the symbol is a number (numerical constant), the mode is A.

e. If the symbol is defined by an EQU, SET, or similar directive, the mode of the symbol is that of the variable field expression in the directive.

The mode of an expression is determined by the following rules:

a. If the expression contains any mode E or C symbol, the expression is mode E.

b. If the expression contains only mode A symbols, the expression is mode A.

c. If the expression contains mode and R symbols, the mode of the expression is R if there is an odd number of mode R symbols. Otherwise, the mode of the expression is A.

The following restrictions apply only to DAS MR and to FORTRAN-compatible output assembly with DAS 8A.

a. No expression can contain symbols of both modes E and C.

b. A mode E expression comprises a single mode E symbol.

c. No mode E, C, or R expression can multiply or divide a mode E or C symbol.

d. No expression can add or subtract a mode C and a mode R symbol, or a mode E and a mode R symbol.

e. No expression can add two or more mode E, C, or R symbols.

f. A mode A symbol can be added to or subtracted from a mode C or R symbol.

Figure 1 illustrates the above rules.

```
EEEE    EXT                 Defines mode E
CCCC    COMN    6           Defines mode C
RTN     ENTR                Defines a symbol (RTN) as a mode R
TBL     BSS     50          TBL is mode R
ABL     BSS     'A'+5       ABL is mode R
LENG    EQU     *-TBL       LENG is mode A (defines area length)
        CALL    EEEE,TBL,LENG
        LDA     *+6         Legal, one-word relative forward
        LDA     CCCC+6      Illegal, one-word not R or A
        LDXI    CCCC+6      Legal, two-word instruction
        LDA     0,1         Legal, loads CCCC+6 in A register
        .
        .
        .
        DATA    EEEE+4      Illegal, value not zero
        DATA    CCCC+4      Legal
        DATA    CCCC+LENG   Legal
        DATA    TBL+LENG 5  Legal, mode is R
```

Figure 1. Manipulation of Expression and Symbol Modes

Relocatability Rules

A relocatable program (DAS 8A, DAS MR) is one that has been assembled with its instruction and directive locations assigned in such a manner that it can be loaded and executed anywhere in memory. When such a program is loaded, the beginning memory address is specified, and a value (known as the relocation bias) is added to the addresses of subsequent relocatable instructions. The programs are usually assembled with a zero relocation bias on the first instruction.

The location counter contains the (relative) address of the instruction or directive currently being executed. The location counter is absolute when it contains the actual address of the instruction, and relocatable when it contains the relative address (the current address of the start of the program).

Symbols can be absolute or relocatable. Expressions, since they contain symbols, can be absolute or relocatable. Constants are always absolute.

The following shows, for each arithmetic operation, whether the result is absolute (abso), relocatable (relo), or illegal.

	A = abso **B = abso**	**A = abso** **B = relo**	**A = relo** **B = abso**	**A = relo** **B = relo**
A + B	abso	relo	relo	illegal
A– B	abso	illegal	relo	abso
A*B	abso	illegal	illegal	illegal
A/B	abso	illegal	illegal	illegal

The relocatable loader can load a program in any area of memory and modify the addresses as it loads so that the resulting program executes correctly. Programs can contain absolute addresses, relocatable addresses, or both. At the beginning of each instruction or data word generated by the assembler, it can be set by the ORG directive. On encountering an ORG directive, the assembler makes the location counter absolute if the corresponding expression is absolute, or relocatable if the corresponding expressions is relocatable.

If a symbol is equated to the location counter, it is relocatable if the location counter is relocatable. Otherwise, the symbol is absolute.

Assembler Input Media

Punched Card Format

Punched cards used as input to the DAS assemblers contain four fields corresponding to the instruction and directive fields:

a. The **label field** is in columns 1 through 6. Its use is governed by the requirements of the instruction or directive.

b. The **operation field** is in columns 8 through 14. It contains the instruction or directive mnemonic. Indirect addressing is specified by an asterisk following the mnemonic.

c. The **variable field** begins in column 16 and ends with the first blank that is not part of a character string. Its use depends on the instruction or directive. If two or more subfields are present, they are separated by commas.

d. The **comment field** fills the remainder of the card. If the variable field is blank, the comment field begins in column 17.

An asterisk in column 1 indicates that the entire card contains a comment.
Note that columns 7 and 15 are always unpunched (blank).

Paper Tape Format

Paper tape used as input to the DAS assemblers contains source statements of up to 80 characters each (not including the carriage return and line feed characters). Each punched statement contains four fields corresponding to the instruction and directive fields. The label, operation, and variable fields are separated by commas, and the comment field starts after the first variable field blank that is not part of a character string. Each statement is terminated by a carriage return (CR) followed by a line feed (LF).

a. **Label field** use is governed by the requirements of the instruction or directive. It is terminated with a comma. If this field is not used, a comma appears as the first character of the source statement.

b. The **operation field** contains the instruction or directive mnemonic. An asterisk following the mnemonic specifies indirect addressing. This field begins immediately following the label field terminator and is terminated by a comma.

c. The **variable field** can be blank, or contain one or more subfields separated by commas. It must immediately follow the instruction field terminator (,). Subfields can be voided by using adjacent commas. This field is terminated by a blank that is not part of a character string, or with a CR or LF.

d. The **comment field** fills the remainder of the statement (from the terminating blank of the variable field to the next CR or LF).

If the first nonblank character of a source statement is an asterisk, the entire statement is a comment.

Assembler Output Listing

DAS produces a source/object listing of the assembled program, as well as a paper tape containing the object program in reloadable format.

The listing can be obtained in whole or in part as the program is being assembled. The source (symbolic) program and the object (absolute) program are listed side by side on the listing device. This device is either a Teletype or a line printer.

The listing is output according to the specifications given by the list and punch control directives in the assembly (DAS 8A, DAS MR).

Error analysis during assembly causes the error messages described below to be output on the line following the point of detection.

The following example illustrates the format of the output listing. A line count appears only on DAS MR listings. The addressing modes are: FORTRAN common reference = C, externally defined = E, indirect pointer = I, and absolute or relative = R.

Address	Code	Mode	Line Count	Symbolic	Source Statement
014000					ORG 014000
014000	000000			ABS	ENTR
014001	001002				JAP* ABS
014002	114000	R			
014003	005211				CPA
014004	001000				JMP* ABS
014005	114000	R			
	000000				END

Error Messages

The assembler checks source statement syntax during both pass 1 and 2. Detectable errors are listed during pass 1. During pass 2, the following information is listed:

a. Error code

b. Location counter value

c. Object code when the instruction is assembled

This information is suppressed by NLIS directives and list-suppression commas in GOTO directives.

The error message appears in the listing line following the statement found to be in error. Each line can hold up to four error messages.

Table 6 lists the DAS error codes and their meanings.

Table 6. DAS Error Codes

Code	Meaning
*AD	Error in an address expression
*DC	Decimal character in an octal constant
*DD	Illegal redefinition of a symbol or the location counter
*E	Incorrectly formed statement
*EX	Illegally constructed expression
*FA	Floating-point number contains a format error
*IL	First nonblank character of a source statement is invalid (the statement is not processed)
*NR	No memory space available for additional entries in assembler tables
*NS	No symbol in the label field of a SET, EQU, MAC, or FORM directive or no symbol in the label or variable field of an OPSY directive, or no symbol in the variable field of a NAME directive
*OP	Undefined operation field (two No Operation (NOP) instructions are generated in the object program; the remainder of the statement is not processed), or illegal nesting of DUP or MAC directives
*QQ	Illegal use of prime (')
*R	Relocatable item where an absolute item should be defined
*SE	Synchronization error: symbol value in pass 2 is different from that found in pass 1
*SY	Undefined symbol in an expression
*SZ	Expression value too large for a subfield, or a DUP directive specifies that more than three statements are to be assembled
*TF	Undefined or illegal indexing specification

(continued)

Table 6. DAS Error Codes *(continued)*

Code	Meaning
*UC	Undefined character in an arithmetic expression
*UD	Undefined symbol in the variable field of a USE directive
*XR	Address out of a range for an indexing specification
* =	Illegal use of a literal

Operating The Assemblers

DAS 4A and 8A Operations

Load the assembler program supplied by Varian into memory using the binary load/dump program (BLD II). Execute it by entering a positive, nonzero value in the A register during loading, or by clearing all registers, pressing (SYSTEM RESET and entering the RUN state. (Set RUN indicator on and press START).

During execution, the program first determines the amount of memory required. It then stores in address 000003 a value one less than the lower limit of BLD II. This is the highest address that the assembler can use without destroying part of BLD II.

DAS 4A and 8A each contain two sections: The I/O section allows the specification of I/O devices for assembler input and output. The second section is the assembler itself.

I/O Section Definitions

The I/O section of DAS 4A and 8A using the Teletype printer, makes three requests for definitions of I/O devices:

```
ENTER DEVICE NAME FOR xx
```

where xx is one of the I/O function names: SI (source input,) LO (list output), or BO (binary output), respectively.

Respond to each request in turn by typing, on the Teletype keyboard, the name of the desired device, followed by a carriage return (CR). Table 7 lists the acceptable device names in response to each request. If the default assignment is desired, merely press CR.

Table7. Acceptable I/O Devices

Assembly Function	Device	Default Assignment
SI (source input)	Teletype paper tape reader: TR Teletype keyboard: TY High-speed paper tape reader: PR Card reader (model 620-22, -23, or -25): CR Magnetic tape: MT nn	TR
LO (list output)	Teletype printer: TY Line printer (model 620-76): LP Line printer (model 620-75): LP1 Line printer (model 620-77): LP2	TY
BO (binary output)	Teletype paper tape punch: TP High-speed paper tape punch: PP Card punch (model 620-27): CP1	TP

If an incorrect device name is typed, the message

```
DEVICE NAME NOT VALID
```

is output and the request repeated.

To terminate the output of any line to the Teletype, press RUBOUT. This error correction feature can be used any time during I/O device specification.

When I/O assignments are complete, the I/O section uses BLD II to load the assembler section into memory.

To restart the I/O section before the assembler section is loaded, set STEP indicator on, clear all registers, press (SYSTEM) RESET, set RUN indicator on and press START.

Assembler Section Definitions

When BLD II relinquishes control to the assembler section, the computer halts with 000001 in the program counter (P register). For an assembler pass 1, set SENSE switch 1; for pass 2, reset SENSE switch 1 and set SENSE switches 2 and 3.

If pass 1 is selected, ready the SI device with the source input media and set RUN indicator on and press START.

For pass 2, ready the SI device with the source input media, ready the BO and LO devices, set RUN indicator on and press START.

The END directive terminates both passes 1 and 2. Pass 1 terminates with 000001 in the P register and 0177777 in the A register. Pass 2 produces the binary object loader text and program listing and terminates when END is encountered with the same register values as pass 1. A MORE directive causes the computer to stop and wait until the SI unit is prepared with the additional source input media, and the RUN state is entered. MORE is indicated by 0170017 in the A register.

The program listing can be suppressed during pass 2 by resetting SENSE switch 2, and the binary output, resetting SENSE switch 3. Error messages cannot be suppressed and are output on the LO device as the error is detected during pass 2.

Synchronization errors (table 6) halt the assembly with 000777 in the A register. To continue the assembly, set RUN indicator and press START. The assembler resets the location counter value to that assigned on pass 1, prints error message ***SE**, and continues the assembly.

Pass 2 can be restarted or repeated for extra copies of the assembled program without repeating pass 1.

At the completion of pass 2, the assembler can accept another assembly using the same I/O devices. For other I/O devices, reload the assembler program, starting with the I/O section.

To restart the assembler, set STEP indicator on, clear all registers, press (SYSTEM) RESET, set RUN indicator on and press START. The assembler halts with 000001 in the P register and is ready to accept another assembly.

The DAS 4A and 8A assemblers can communicate with any one of the magnetic tape transports on a controller. Up to four transports may be connected to each of the magnetic tape controllers. A configuration may have one to four magnetic tape controllers.

The magnetic tape transport number and controller device address is specified in the device name specification of the I/O Control Section based upon the following table:

Device Name	Address (in octal)	Transport Number
MT00	010	1
MT01	010	2
MT02	010	3
MT03	010	4
MT10	011	1
MT11	011	2
MT12	011	3
MT13	011	4

Device Name	Address (in octal)	Transport Number
MT20	012	1
MT21	012	2
MT22	012	3
MT23	012	4
MT30	013	1
MT31	013	2
MT32	013	3
MT33	013	4

DAS MR Operations

Since DAS MR operates under MOS and uses the MOS I/O control system, the I/O devices can be defined as required (refer to MOS section of this handbook).

DAS MR inputs the symbolic source statements from the processor input (PI) logical unit in alphanumeric mode, and outputs them in the same mode on the processor output (PO) logical unit. When DAS MR detects the END directive, it terminates pass 1, returns to the beginning of the source program, and begins pass 2. During pass 2, the source statements are the input from the system scratch (SS) logical unit, a listing is output on the LO unit, and the binary object program is output on the BO unit. Note that PO and SS must be the same magnetic tape, drum, or disc unit.

For an assembly without a program listing, input the following directive to the MOS executives when requesting the assembly:

```
/ASSEMBLE        N
```

For a binary object program, input

```
/ASSEMBLE        B
```

If the memory map portion (symbol table, external names, and entry names) is not wanted, input

```
/ASSEMBLE        M
```

To read the same physical symbolic source statements for both assembly passes, input

```
/ASSIGN          PO=DUM, SS=PI
/ASSEMBLE
```

The processor output listing serves as a copy of the program; it can be input for another assembly.

With an operating system the DAS MR user gains the facilities provided in either MOS or VORTEX. The features of MOS are described in detail in a later section in this handbook.

The standalone system is operated with procedures also used for the standalone FORTRAN system (described in a later section).

Binary Loader Programs

TABLE OF CONTENTS

BINARY LOADER PROGRAMS

Two stand-alone loader programs are available for the Varian 73 and 620 computer systems: **Binary Load/Dump (BLD II)** and **Binary Card Loader (BCL I)**. The BLD II program prepares the computer for the loading of non-relocatable object programs from a high-speed or Teletype paper tape reader. It also allows a program stored in memory to be punched on paper tape in re-loadable format. For computer systems using card I/O devices, the BCL I program loads binary information from either a model 620-22 or 620-25 card reader. No memory dump feature is included in the BCL I program.

SECTION 1
BINARY LOAD/DUMP PROGRAM (BLD II)

BLD II is loaded using the bootstrap loader routine, which specifies the input reader. Once loaded, BLD II automatically relocates itself into the upper part of the highest 4K memory increment, unless the operator specifies another 4K increment. BLD II also dynamically adapts itself to load object program tapes from the input device specified in the bootstrap loader routine, and performs a check-sum of object program records.

After BLD II has been loaded into memory, it need not be reloaded for the entering of subsequent object programs.

Initially, BLD II occupies addresses 007000 through 007755 of the first 4K memory increment, where it does not interfere with the bootstrap loader routine occupying addresses 007756 through 007776. Immediately after loading, BLD II relocates to occupy addresses 0x7400 through 0x7755, where x denotes the highest, or operator specified, 4K of memory.

x =	Memory Increment
0	4K
1	8K
2	12K
3	16K
4	20K
5	24K
6	28K
7	32K

Entry to BLD II to load object program tapes is always 0x7600, and entry to punch binary object tapes of memory contents is 0x7404.

LOADING THE BOOTSTRAP ROUTINE

Under normal conditions the bootstrap loader routine would be loaded automatically as follows:

a. With the POWER switch in the ON position, place the computer in the run mode by pressing the STEP/RUN switch (RUN indicator is blinking).

b. Insert the BLD II tape in the reader with the first binary frame at the read station.

c. Press the boot switch (RUN indicator is now on). This transfers the bootstrap program from the processor's control store to computer memory and executes loading of the BLD II program.

For maintenance purposes it may be desirable to load the bootstrap routine manually.

Table 1-1 lists the manual bootstrap loader routines. If the high-speed paper tape reader is to be used for subsequent program loading, select the column headed High-Speed Reader Code; for the Teletype paper tape reader, select the column headed Teletype Reader Code.

To load the bootstrap loader routine:

a. Ensure that computer power is turned on and that the system is initialized.

b. Load the starting memory address of the bootstrap loader (007756) into the P register.

c. Press MEM switch momentarily.

d. Clear the console display (Press DISPL CLR).

e. Select the first bootstrap loader instruction from the appropriate column in table 1-1, and load it into the console display.

f. Press ENTER to load the display contents into the address specified by the P register, which is incremented by one after the instruction is loaded.

g. Clear the display (Press DISPL CLR).

h. Repeat steps d, e, f, and g for each bootstrap loader instruction.

Table 1-1. Bootstrap Loader Routines

Address	High-Speed Reader Code	Teletype Reader Code	Symbolic Coding		
007756	102637	102601	READ	CIB	RDR
007757	004011	004011		ASLB	NBIT – 7
007760	004041	004041		LRLB	1
007761	004446	004446		LLRL	6
007762	001020	001020		JBZ	SEL
007763	007772	007772		(Memory address)	
007764	055000	055000		STA	0,1

(continued)

Table 1-1. Bootstrap Loader Routines (continued)

Address	High-Speed Reader Code	Teletype Reader Code		Symbolic Coding	
007765	001010	001010		JAZ	LHLT+1
007766	007000*	007000*		(Memory address)	
007767	005144	005144		IXR	
007770	005101	005101	ENTR	INCR	1
007771	100537	102601		EXC**	RDON
007772	101537	101201	SEL	SEN	IBFR,READ
007773	007756	007756		(Memory address)	
007774	001000	001000		JMP	* – 2
007775	007772	007772		(Memory address)	

NOTE

The bootstrap loader routine is always loaded into the specified addresses of the first 4K memory increment, regardless of available memory.

* Replace this code with 007600 if the test executive of MAINTAIN II (refer to document number 98 A 9952 06R) is to be loaded and executed.

** CIB instruction if TTY bootstrap.

To verify bootstrap loading:

a. Initialize the system by pressing (SYSTEM) RESET.

b. Load 007756 into the P register.

c. Select the memory for display by pressing MEM and press DISPL.

The contents of the memory addresses are displayed sequentially each time the DISPL switch is pressed. If an error is found, load the correct instruction code into memory. **Note that the P register error address is always the error address plus one.**

BLD II, and subsequent object programs, can now be loaded into memory.

LOADING THE BLD II PROGRAM

CAUTION

To adapt to the input device, BLD II examines address 000200 to determine if the system includes the automatic bootstrap loader (ABL) option, then the

contents of the first address of the manual bootstrap loader routine, both of which can indicate the input device. If address 000200 inadvertently contains one of the two input device codes, and the device used is different, BLD II malfunctions.

After the bootstrap loader routine has been successfully loaded into memory:

a. Clear the instruction register.

b. Load 007770 into the P register.

c. Load 007000 into the X register.

d. Set the SENSE switch(es) for the desired program option (table 1-2).

e. Turn on the paper tape reader specified by the bootstrap loader routine.

f. Position the BLD II program tape in the reader with the first data frame after the position-8-only punches (figure 1-1) under the high-speed reader head or under the reading station of the Teletype reader.

g. To load tape, press RUN, then START. Loading is complete when the computer changes to step mode.

Table 1-2. BLD II SENSE Switch Options

SENSE Switch	When Set =
1	Allows selection of any 4K memory increment in which BLD II is to operate, or specification of a nonstandard device address for the high-speed paper tape punch. After BLD II is loaded, the computer halts with 07014 in the P register. To specify a 4K memory increment, load one of the following in the A register: **A Register** — **Memory Increment** 000000 — First 4K 000001 — Second 4K 000002 — Third 4K 000003 — Fourth 4K 000004 — Fifth 4K 000005 — Sixth 4K

(continued)

Table 1-2. BLD II SENSE Switch Options *(continued)*

A Register	Memory Increment
000006	Seventh 4K
000007	Eighth 4K

The standard high-speed paper tape punch device address is 037. To specify a nonstandard device address, load it into the B register.

Result: Pressing START initiates the relocation of BLD II from the first 4K memory increment and implements the punch address. The computer halts with zeros in the A, B, and X registers and 0x7600 in the P register, where x = the specified increment as described above. Object program tapes can then be loaded.

SENSE Switch	When Set =
2	Adjusts the program for Teletype paper tape punch output. (For use when input is from high-speed reader, but a high-speed punch is not available.) **Result:** BLD II and the object program can be loaded and executed without further operator intervention.
3	Allows splicing an object program to the BLD II program tape.

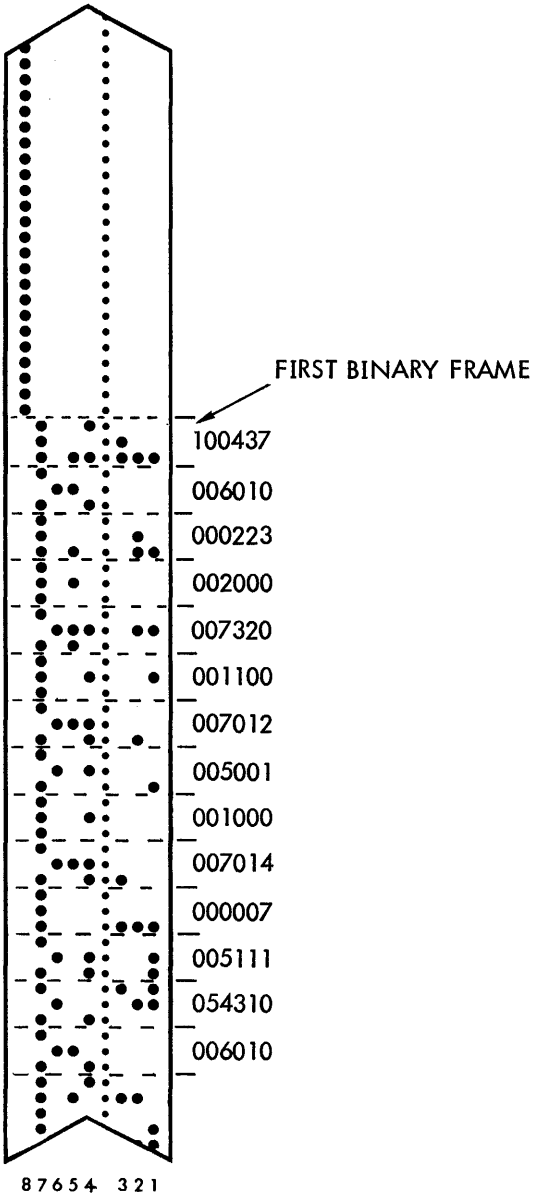
NOTE

If no SENSE switches are set, the BLD II program is loaded and relocates automatically to the highest 4K memory increment. The computer then halts with the entry address for reading object program tapes in the P register (0x7600) and zeros in the A, B, and X registers.

If SENSE switch 1 was set:

a. Reset SENSE switch 1.

b. Clear the A register.



VTII-1889

Figure 1-1. BLD II Tape Format (Bootstrap-Loadable)

c. Load the appropriate values, as defined in table 2-2, in the A and/or B registers.

d. Press START.

When BLD II loading is complete, the computer halts with 0x7600 in the P register unless SENSE switch 3 was set (table 1-2), in which case the computer implements loading and execution of the spliced object program.

Remove the BLD II program tape from the reader after loading, and reset SENSE switch 2, if applicable.

LOADING AN OBJECT PROGRAM

Object programs can be loaded from the bootstrap-routine-specified device immediately after BLD II. For all subsequent loadings, make sure that the P register is set to 0x7600.

Verification

To ensure that an object program tape contains no errors before it is loaded into memory, BLD II has a check-sum error-checking option. To use this option:

a. Turn on the bootstrap-routine-specified reader.

b. Position the object program tape in the reader with leader at the reading head (figure 1-2).

c. Load minus value (0100000) into the A register.

d. Clear the instruction register.

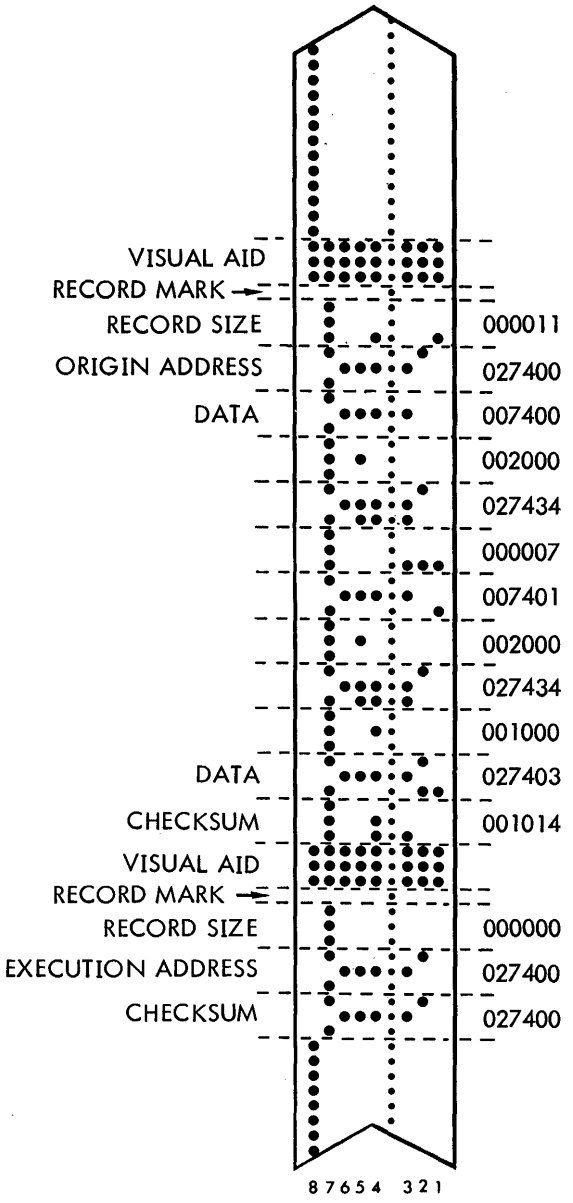
e. Set RUN indicator on and press START.

No errors are indicated by the computer halting with:

P register = 0x7600
A register = 0100000
B register = 000000
X register = execution address

If a check-sum error occurs, the computer halts with:

P register = 0x7600
A register = 0100000
B register = 0177777
X Register = Address of last record



VTII-1888

Figure 1-2. Object Program Tape Format

To retry a check-sum error record, reposition the object program tape at the previous visual aid and press START. If a check-sum error is again read, visually check each character in the record for an error in punching or damaged tape.

Load Program and Halt

To load the object program and halt before execution:

a. Turn on the reader and position the tape in the reading station.

b. Clear the A, B, X, and instruction registers.

c. Load 0x7600 into the P register.

d. Set RUN indicator on and press START.

Correct loading is indicated when the computer halts with:

P register = 0x7600
A register = 000000
B register = 000000
X register = execution address

A check-sum error is indicated by the conditions described for object program tape verification described above.

Load Program and Execute

Programs can be loaded and immediately executed using the steps described above for the load-and-halt option, except in step b load 000001 (or any positive number) in the A register.

PUNCHING PROGRAM TAPES

The BLD II program adapts to the input reader and the output punch devices by interrogating the bootstrap loader routine. Setting SENSE switch 2 (table 1-2) prior to loading BLD II program adjusts the program for Teletype punch output regardless of the bootstrap-routine-specified devices.

To punch reloadable object program tapes after the programs have been loaded into memory, turn on the punch and:

a. Load the beginning address of the area to be punched into the A register.

b. Load the final address to be punched into the B register.

c. Load the first instruction to be executed at load time into the X register,

   OR

   if noncontiguous memory areas are to be punched, load minus one (177777) into the X register.

d. Load 0x7404 (entry address to BLD II to punch object tapes) into the P register.

e. Clear the instruction register.

f. Press (system) RESET, set RUN indicator on and press START.

The program punches the object tape and the computer halts with all registers unaltered.

If noncontiguous areas are to be punched, perform steps a through f. Prior to punching the last area, load the first instruction to be executed at load time into the X register.

PUNCHING MEMORY CONTENTS

To punch a tape of the binary memory contents on the high-speed paper tape punch, SENSE switch 2 must not be set when BLD II is loaded. To punch a tape from memory on the Teletype punch, SENSE switch 2 must be set (if the input reader is a high-speed paper tape device).

The operator can specify that tapes be punched in binary format for reloading using the BLD II, or that the BLD II program be punched in bootstrap-loadable format.

To punch a tape in binary format, use the procedures described above for punching program tapes.

To punch a bootstrap-loadable tape of BLD II itself:

a. Load 0x7400 into the P register.

b. Clear the A and B registers.

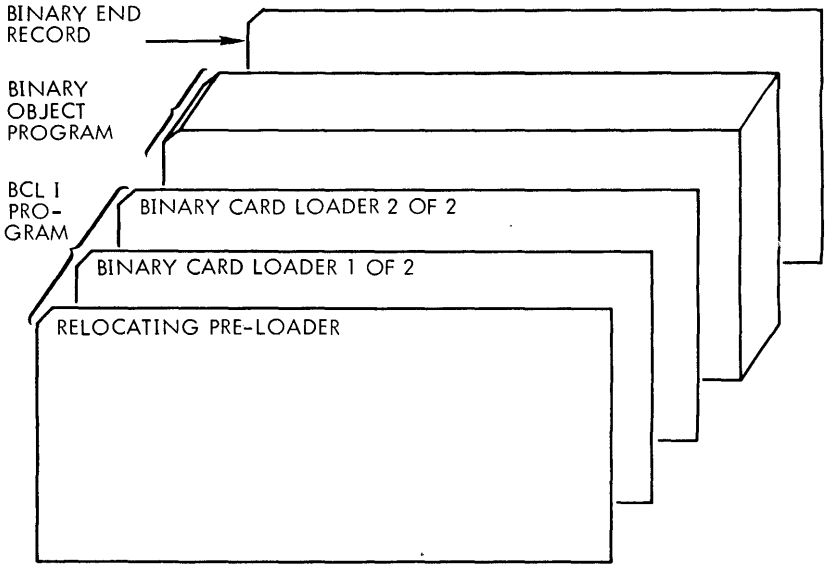
c. Load a nonzero value into the X register.

d. Press (system) RESET, set RUN indicator on and press START.

SECTION 2
BINARY CARD LOADER (BCL I)

The BCL I program loads and executes card object programs with a minimum of operator involvement. The program automatically allocates and positions the card loader routine in an area at the top of an operator-specified memory module (16-bit word only). A job stream containing the BCL I cards is shown in figure 2-1.

A bootstrap routine (table 2-1) performs the initial loading of the BCL I into main memory, and then passes control to the relocating pre-loader portion of the BCL I program. The relocating pre-loader loads the binary card loader into main memory and relocates it in an area of memory for permanent residency. The relocating pre-loader then passes control to the binary card loader, which in turn reads binary information from 16-bit data words on cards and transfers it to memory.



VTII-1898

Figure 2-1. Job Stream Containing BCL I Card

BOOTSTRAP ROUTINE

A BCL I bootstrap routine (table 2-1) loads the BCL I program from punched cards into memory and automatically initiates execution of the relocating pre-loader.

RELOCATING PRE-LOADER

The relocating pre-loader determines the highest address of physical memory and computes a relocation address based on the upper-boundary address. If the highest memory address is specified by the operator, the pre-loader will compute the relocation address based on that value. After relocating the BCL to the memory area, the pre-loader automatically transfers control to the BCL routine.

BINARY CARD LOADER

The binary card loader loads object data from cards produced by DAS 4A or DAS 8A assemblers. The data formats for the BCL I and object program cards are shown in figures 2-2 and 2-3, respectively. In the BCL I format, each 16-bit word is contained in two columns of rows 2 through 9. In the object program format, the 16-bit words are arranged serially beginning with row 12 of column 1. The first 16 bits on an object-program card contain a count of the object words on the card; the second 16 bits contain the load address. Object words begin in the third 16 bits. As each object word is loaded the word count is decremented. A 16-bit checksum, the last entry on the card, is compared with the checksum computed by the loader. Any discrepancy causes the computer to stop indicating a checksum error. An end record has its first 16 bits all ones. The loader assembles the second 16 bit on the end-record card as an execution address. The end-record checksum is disregarded and control passes to the loaded object program at the point specified by the execution address.

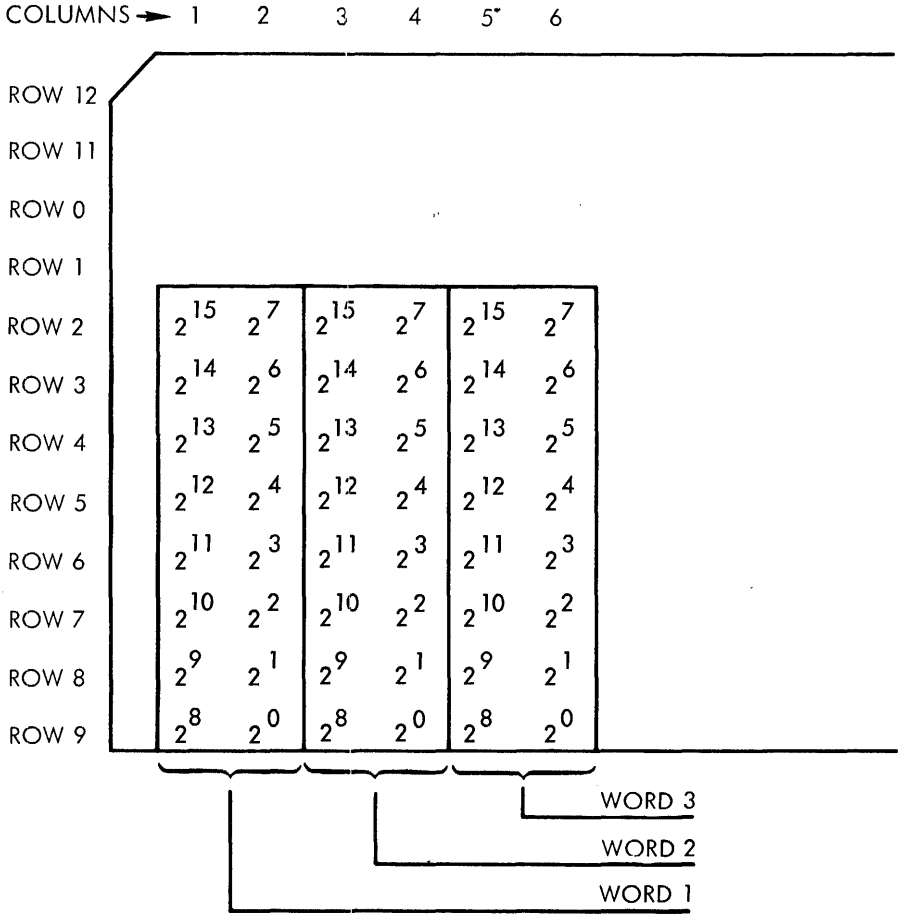
Table 2-1. BCL I Bootstrap Routine

		Symbolic Instruction			
Address	**Octal Code**	**Label**	**Operation**	**Variable**	**Comment**
000114	102530	BOOR	CIA	030	Input card column
000115	004250		LRLA	8	Position to high order
000116	101130		SEN	0130,BOOS	Character ready
000117	000122				
000120	001000		JMP	*-2	Wait until ready

(continued)

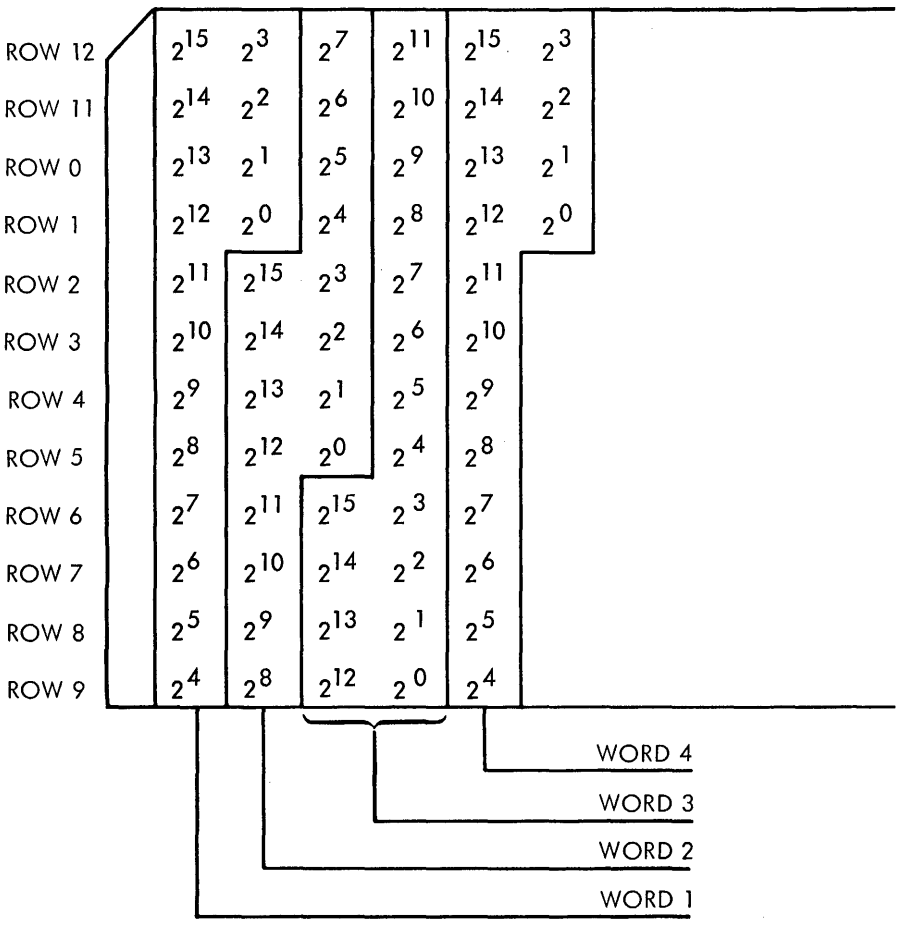
Table 2-1. BCL I Bootstrap Routine (continued)

Address	Octal Code	Symbolic Instruction Label	Operation	Variable	Comment
000121	000116				
000122	102130	BOOS	INA	030	Low order 8 bits
000123	055000		STA	0,1	Store word
000124	005144		IXR		Increment store pointer
000125	001000		JMP	BOOU	Do again
000126	000131				
000127	000000	BOOT	DATA	PLD	
000130	100230		EXC	0230	Read a card
000131	101130	BOOU	SEN	0130,BOOR	Character ready
000132	000114				
000133	101630		SEN*	0630,BOOT	End of card, reader now ready
000134	100127				
000135	001000		JMP	*-4	
000136	000131				
000137	000000	RLOD	DATA	0	Loader start address



VTII-1890

Figure 2-2. Format of BCL I Card



WORD 1 COUNT OF OBJECT WORDS, OR ALL ONES IF AN END RECORD
WORD 2 LOAD ADDRESS, OR TRANSFER ADDRESS IF AN END RECORD
WORD 3 FIRST OBJECT WORDS
WORD 4 SECOND OBJECT WORD
THE LAST WORD IS A CHECKSUM (EXCLUSIVE OR)

VTII-1891

Figure 2-3. Format of Object Program Cards

OPERATING PROCEDURE FOR BCL I

a. Using the computer control panel, load the bootstrap routine (table 2-1) into memory.

b. Load 000130 into the P register

c. Load zero into the X register

d. Load the B register with
   1. zero if BCL I is to reside at the top of main memory, or
   2. the octal value for the upper boundary of the memory module in which the loader is to reside. The accepted values are

B-Register Value (octal	Desired 4K Memory Module	Memory Boundary (decimal)
010000	1	4,096
020000	2	8,192
030000	3	12,288
040000	4	16,384
050000	5	20,480
060000	6	24,576
070000	7	28,672

e. Place the 3-card BCL I object program in the card reader followed by the binary object deck to be loaded (figure 2-1).

f. Ready the card reader

g. Start the computer

Note: the contents of the A register are not significant.

RE-USING BCL I

When the procedure outlined above is followed, BCL I resides in the upper 80 words of the 4K memory module in which it is relocated. The procedure for operation of the stored BCL I routine is:

a. Load 0x7660 into the P register, where x is 0 through 6 designating the 4K memory module in which the loader resides.

b. Place the binary object deck in the card reader

c. Ready the card reader

d. Start the computer

ERROR INDICATION

The only error condition indicated by BCL I is a checksum error. This error causes the computer to halt with the P register set to 0x7767 and the instruction register set to 000525. For retry, re-feed the last card read. A repetition of the error suggests a faulty assembly of the DAS 4A or DAS 8A.

Debugging Program (AID II)

TABLE OF CONTENTS

AID II DEBUGGING PROGRAM

AID II DEBUGGING PROGRAM

The **Varian 73/620 AID II Debugging Program** is available with Varian 73 and 620 systems to provide on-line program checkout and correction. By entering AID II commands on the Teletype keyboard, the operator can:

a. Display and alter the contents of registers and any memory address or group (block) of addresses.

b. Transfer (trap) into or out of selected blocks of memory and search for specific conditions.

c. Load, monitor, and alter any program.

As an added feature, data can also be transferred (dumped) from memory to magnetic tape, punched out on paper tape, or printed on the Teletype printer. Object programs can thus be converted from one media to another, simply and directly.

AID II is loaded into computer memory using the binary load/dump program (BLD II). Once loaded, AID II resides in memory addresses 0x6000 through 0x7377, where x denotes the highest available 4K memory increment, as follows:

x =	Memory Increment
0	4K
1	8K
2	12K
3	16K
4	20K
5	24K
6	28K
7	32K

The programmer is responsible for ensuring that a program to be debugged does not interfere with those areas of memory containing BLD II and AID II.

Loading AID II

To load AID II into memory:

a. Ensure that the bootstrap loader routine and BLD II are correctly loaded.

b. Turn on the reader used to load BLD II and position the AID II program tape with leader at the reading station.

c. Clear the B, X, and instruction registers, and load 000001 into the A register.

d. Load 0x7600 into the P register (i.e., the BLD II entry address for loading program tapes; refer to BLD II section for the definition of x).

e. Place the computer in the run mode.

Loading is complete when the program outputs a carriage return (CR) and line feed (LF) and rings the Teletype bell.

Programs to be debugged can be loaded either before or after AID II loading.

Register and Memory Modification

With AID II and the program to be debugged entered, the computer in run mode, and the Teletype operating on-line, the Teletype keyboard entries summarized in table 1 produce the indicated results.

The pseudoregisters referred to in the following descriptions denote software buffers that duplicate the actual contents of the computers's operation registers. A command to change register contents, in effect, changes the specified pseudoregister contents, which are then transferred to the corresponding operation register.

Table 1. AID II Register/Memory Modification Commands

Command	Operation
A **B** **X**	Displays (prints) the contents of the indicated pseudo-register on the Teletype printer. To change the contents, type the desired octal number and a period; otherwise, type only a period.
Cx	Displays (prints) the contents of memory address x on the Teletype printer. To change the contents, type the desired octal number, followed by a period to execute the command or by a comma to request display of the next sequential address contents. Otherwise, type only a period.

(continued)

Table 1. AID II Register/Memory Modification Commands *(continued)*

Command	Operation
Gx.	Loads the contents of the pseudoregisters into the respective A, B, and X registers and starts program execution at address x.
Ix,y,z,.	Stores the value of z in all memory addresses starting at address x and ending at address y.
Sx,y,z,m.	Searches through memory starting at address x and ending at address y for the value of z masked by the value of m. A masked-search compares the value of z with each bit corresponding to a one in the m value. Each time the values compare, the address and value are printed on the Teletype printer. If an **N** is typed instead of a mask value, the program searches for the negative value of z. Omission of m assumes an all-ones mask.
Ty,x.	Transfers execution of an operational program to address y when the program reaches the instruction in address x. This trapping feature permits interrupting a program sequence without internal patching. The program also displays the transfer address and the contents of the A, B, and X pseudoregisters, respectively.
Ty,.	Continues trap from last break point.
Vx.	Displays the contents of memory on the Teletype printer beginning at address x, continuing until a RUBOUT character is typed. The display (dump) is printed in columns: the left column is the octal base address, and the contents of eight memory addresses, in ascending order, appear in the next eight columns. The first number in succeeding lines indicates the base address for the next eight memory address contents.

Usage Examples

NOTE

In the following examples, operator inputs are represented in bold type. Other entries are program responses output to the Teletype printer.

Display the contents of a pseudoregister:

```
A   142340   .
B   001000   .
X   006003   .
```

Display and change the contents of a pseudoregister:

```
A   010454   10406.
B   006016   10406.
X   007413   10406.
```

Display the contents of memory address 002050:

```
C2050        =   102401   .
```

Display and change the contents of memory address 002050, then display the next two addresses:

```
C2050        =   102401   103402,
( 002051 ) = 000067   ,
( 002052 ) = 177777   .
```

Display memory contents starting at address 006000:

```
V6000.
( 006000 )  010454  002000  ...
( 006010 )  005145  004543  ...
( 006020 )  005041  001000  ...
( 006030 )  006217  001000
```

NOTE

When displaying memory contents, eight columns of data actually follow the base address in the first column. Space limitations prohibit an actual representation herein.

(Display terminated by entering RUBOUT.)

Execute the program beginning at address 000500:

G500.

Store 0177777 in memory addresses 000200 through 000210:

I200,210,177777,.
1200,210,-1

Search memory addresses 000200 through 000240 for a content of 0106213 masked by 0177777 and display addresses that compare:

S200,240,106213,177777.
(000220) = 106213
(000235) = 106213

Trap to memory address 000204; start execution from address 000100; and display the trap address and the A, B, and X register contents if the trap is reached. **If not, reload the original contents into both trap locations.**

T204,100.
(000204) 142340 002000 010405

Paper Tape Handling

The Teletype paper-tape reader and punch can be controlled through AID II to read object program tapes into, and punch program tapes from, computer memory.

With AID II entered, the computer in run mode, and the Teletype and its paper tape system operational, the Teletype keyboard entries summarized in table 2 produce the indicated results.

Table 2. AID II Paper Tape Commands

Command	Operation
Dx,y,z,.	**Punches a program tape from the contents of address x** through address y, specifying execution address z.
Lm.	Reads an object program paper tape into memory. If the value of m is 1 and no check-sum errors are encountered, the program is executed.

(continued)

Table 2. AID II Paper Tape Commands (continued)

Command	Operation
	If the value of m is 0 and no check-sum errors are encountered, the contents of the A, B, and X registers, respectively, are output on the Teletype printer: A register = 000000, B register = 000000, and X register = execution address.
	If m is − 1, the operation is the same as zero except the object tape is verified but not loaded into core.
	If the program detects a check-sum error, the printout is the same as m = 0 except B register is = to A 0177777 and X register is = to the address of last record read correctly.

Note

AID II utilizes BLD II to effect loading and punching. For proper operation, BLD II must reside in the same 4K increment of memory as AID II.

Magnetic Tape Handling

Data can be manipulated from and to magnetic tape through AID II commands.

With AID II entered, the computer in run mode, the Teletype keyboard on-line, and the selected magnetic tape unit operational, the Teletype keyboard entries summarized in table 3 produce the indicated results.

In the following descriptions, x specifies the magnetic tape controller device address coded as 0, 1, 2, and 3, where 0 = first system magnetic tape controller, 1 = second system controller, etc. Note that each magnetic tape controller monitors up to four magnetic tape units and that AID II communicates only with the first unit on each controller.

Table 3. AID II Magnetic Tape Commands

Command	Operation
Ex.	Writes a file mark on the specified unit tape.
Fn,x.	Skips to file n on the specified unit tape.
N.	Skips to the next file on the previously designated unit tape.
Px.	Backspaces one record on the specified unit tape.
Rx	Reads an object magnetic tape into memory from the specified magnetic tape unit. Terminating the command with a period causes the program to be loaded and control returned to AID II. If the command is terminated with a comma, the program is loaded and executed. If AID II outputs an uparrow (↑) on the Teletype printer, a file mark was read on the tape. The output of an octal number indicates the address of a parity error.
Wa,b,c,x.	Writes an object magnetic tape from memory, starting at address a and ending at address b with an execution address of c, on the specified magnetic tape unit.

Error Message and Correction

If an AID II command is input incorrectly, AID II terminates further input by outputting a CR and LF and ringing the Teletype bell. An example of incorrect input is an attempt to type a nonoctal number (i.e., a decimal 8 or 9). Note that octal numbers need not be preceded by a zero. To recover, correctly retype the entry.

An input command can be aborted before termination by the backslash (\) character.

Magnetic and paper tape error discriptions are included in tables 2 and 3.

Source Program Editor (EDIT)

TABLE OF CONTENTS

SOURCE PROGRAM EDITOR

SOURCE PROGRAM EDITOR

Varian's 73/620 source program editor (EDIT) allows the computer programmer to create and modify symbolic source programs on paper tape. Source programs can be loaded directly into computer memory from an on-line Teletype keyboard, listed with identifying line numbers on the Teletype printer, and modified using EDIT commands input from the Teletype keyboard.

Source programs already formatted on paper tape can be loaded into memory, listed, modified with EDIT generating a paper tape of the modified program ready for assembly or compilation.

An added feature of EDIT is its ability to search through the source program and point to a specific character or group of characters, as well as entire lines and groups of lines.

EDIT has two modes of operation: command and text. In command mode, EDIT accepts inputs from the Teletype keyboard specifying the EDIT function and, optionally, line numbers and searching parameters. In text mode, characters typed on the Teletype keyboard or read from paper tape are stored in a text buffer for subsequent manipulation and/or output. The text buffer represents available memory, i.e., those memory addresses not occupied by the bootstrap loader routine, the binary load/dump program (BLD II), and the EDIT program routines.

In text mode, EDIT runs without an operating system. Both MOS and VORTEX include editing functions which are an alternative in their environments.

EDIT operates in the minimum configuration of a computer system (4K to 32K of memory) and 33/35 ASR Teletype. However, EDIT determines the size of memory and uses of all available memory for the editing buffer; only the binary loader at the top of memory is served. Use of the high-speed paper tape reader and/or punch for input/output is optional.

Loading EDIT

To load the EDIT program into memory:

a. Ensure that the bootstrap loader routine and BLD II are correctly loaded.

b. Turn on the reader used to load BLD II and position the EDIT program tape with leader at the reading station.

(continued)

c. Clear the B, X, and instruction registers.

d. Load 000001 into the A register.

e. Load 0x7600 into the P register (i.e., the BLD II entry address for loading object program tapes).

f. Place computer in the run mode.

Loading is complete when the EDIT program outputs, on the Teletype printer, the message:

```
SOURCE PAPER TAPE PROGRAM

INPUT DEVICE (H OR T)
```

If the high-speed paper tape system is to be used for text input to EDIT, type H on the Teletype keyboard, and type T if the Teletype is the input device. The program then outputs

```
OUTPUT DEVICE (H OR T)
```

Respond as described above for defining the input device. EDIT dynamically adapts to use the specified equipment and enters the command mode, outputting a carriage return (CR) and line feed (LF), followed by an asterisk (*), to the Teletype printer.

Once entered, EDIT can be restarted at any time by clearing all registers and pressing RUN or START.

NOTE

To change input and output devices from those initially specified, EDIT must be reloaded using the procedures described above.

EDIT Commands

With EDIT loaded, the computer in run mode, and the Teletype operating on-line, the Teletype keyboard entries summarized in table 1 produce the indicated results. Pressing the RETURN key terminates and executes all EDIT commands.

Table 1. EDIT Commands

Command	Operation
A	Enter text mode and add the following text input from the Teletype keyboard to the contents of the text buffer.
nC	Delete the line specified by n, and replace it with new text.
m,nC	Delete and replace lines m through n.
nD	Delete line n.
m,nD	Delete lines m through n.
F xxxx	Search the entire contents of the text buffer for character string xxxx (maximum number of characters, 72). Output sequential text lines until the string is detected and the line on which it appears is output. If the string is not found, return to command mode, and output CR, LF, and *.
nF xxxx	Go to line n and search it and succeeding lines for character string xxxx (see above).
G	List (output on the Teletype printer) the next sequential line whose first character is alphabetic.
nG	Go to line n and list the next line whose first character is alphabetic.
I	Insert the following text before the first line in the text buffer.
nI	Insert the following text before line n.
K	Delete the entire contents of the text buffer.
L	List the entire contents of the text buffer, assigning sequential line numbers (decimal), on the Teletype printer.
nL	List line n.
m,nL	List lines m through n.

(continued)

Table 1. EDIT Commands *(continued)*

Command	Operation
P	Punch the contents of the text buffer on paper tape using the output device specified at edit loading time.
nP	Punch line n.
m,nP	Punch lines m through n.
R	Read (append) the following text input from the device specified EDIT loading time to the contents of the text buffer.
S	Search the contents of the text buffer for the character input after RETURN. Output sequential text lines on the Teletype printer until the line in which the character appears is printed. If the character is not found, return to command mode, and output CR, LF, and *.
nS	Go to line n and search for the character input after RETURN (see above).
m,nS	Search lines m through n for the character input after RETURN (see above).
T	Punch approximately 20 inches of leader/trailer on paper tape using the output device specified at EDIT loading time.

NOTES

Line numbers when specified in EDIT commands are decimal integers derived from the output of a listing command. The value of n must be greater than that of m.

Execution of **all** EDIT commands begins when the RETURN key is pressed.

Table 2 lists EDIT functions that are controlled by the use of Teletype special-purpose keys. Note that their use differs in the two modes of operation.

Table 2. Teletype Key EDIT Functions

Teletype Key	Command Mode	Text Mode
RETURN	Execute the instruction	Load the input line into the text buffer
←	Illegal	Delete one character to the left and output
RUBOUT	Cancel the instruction	Delete all the line to the left and output \
CTRL and C (simultaneously)	Remain in instruction mode and output an asterisk (*)	Return to instruction mode and output an asterisk (*)
. (period)	Current line number (used alone or with the minus sign and number, e.g., 1. −8 refers to the eighth line preceding the current line)	Legal text character
/ (slash)	Number of the last line in the text buffer	Legal text character
=	Used with . and / to obtain their values	Legal text character
ESCAPE (ESC)*	List the next line	Ignored
CTRL and TAB (simultaneously)	Illegal	Interpreted as seven spaces on the Teletype printer output

* On the Model 35 Teletype, simultaneously press SHIFT, CTRL, and K.

Usage Example

To illustrate the use of EDIT commands and Teletype key functions, assume we wish to search line 20 for the character A and replace it with the character X. Note that the

Teletype keys are shown enclosed in parentheses where they are applicable and that the simultaneous pressing of two or more keys is illustrated as follows: (SHIFT)(CTRL)(K).

a. To ensure that EDIT is in command mode, type

```
(CTRL)(C)
```

b. EDIT responds with a CR, LF, and *. Type

```
20S(RETURN)
```

c. EDIT enters a delay loop and waits for input of the character for which it is to search. Type

```
A(RETURN)
```

d. EDIT goes to line 20 and types it until an A is found:

```
XYZ   LDA
```

then waits for input. Type

```
←X(RETURN)
```

Other editing options available for use in step d are:

a. To delete the line to the left, type RUBOUT.

b. To delete the line to the right, type RETURN.

c. To delete the entire line, type the appropriate deletion command (table 1).

d. To delete characters from right to left, type ← once for each character.

Error Messages

EDIT checks all commands input to it for valid parameters and correct formatting. When an error is detected, EDIT:

a. Types a question mark on the Teletype printer.

b. Issues a CR and LF.

c. Types an asterisk.

d. Waits for a valid command.

The following conditions are recognized as errors:

a. Incorrect response to the I/O device queries at loading time.

b. A nonexistent command code.

c. Commands terminated with any character other than RETURN.

d. A starting line number that is greater than an ending line number.

e. Transposition of command parameters.

f. Specifying a line number whose value is greater than the last line in the buffer.

g. A deletion command that does not specify a line number.

h. Pressing the ESCAPE (ESC) key to list the next line in the buffer when the buffer is empty.

When text being loaded into the text buffer exceeds the capacity of the buffer, EDIT outputs the message

```
BUF FULL
```

and returns to command mode. To save the buffer contents and continue processing:

a. Type a punch (**P**) command (table 1) and RETURN.

b. After punching is complete, restart EDIT by clearing all registers and pressing START.

The following options are also available:

a. List, modify, and punch the buffer contents before restarting EDIT.

b. Abort the current source program edit and continue processing with a new program.

Mathematical Subroutines

TABLE OF CONTENTS

MATHEMATICAL SUBROUTINES

MATHEMATICAL SUBROUTINES

In support of Varian 73/620 computer applications programs that require mathematical computations, Varian provides a comprehensive **Mathematical Subroutine Library** with complete, easily accessible subroutines.

The mathematical subroutines are grouped into four major categories: fixed-point arithmetic, floating-point arithmetic, arithmetic functions (both real and complex), and number and character conversions. The subroutines are called by other programs and fill the mathematical requirements of virtually all computer applications.

The mathematical subroutine library is described in detail in the Varian 620 Subroutine Descriptions Manual (document number 98 A 9902 044).

Fixed-Point Arithmetic

The fixed-point arithmetic subroutines are for applications that demand a high-speed arithmetic package. They include:

a. Addition, subtraction, multiplication, and division (single- and double-precision)

b. Two's complement (double-precision)

c. Absolute value

d. Transfer of sign

Fixed-point, single-precision multiplication (**XMUL**) provides a software version of the multiplication hardware. XMUL uses successive addition of the multiplicand with appropriate left-shifts.

Fixed-point, single-precision division (**XDIV**) provides a software version of the division hardware. XDIV uses an unsigned, nonrestoring division algorithm.

Fixed-point, double-precision addition (XDAD) adds the double-precision number whose address is in the calling sequence to the double-precision number in the A and B registers. The low-order halves of the numbers are added first, and, if there is a carry, it is added to the high-order sum.

Fixed-point, double-precision subtraction (XDSU) subtracts the double-precision number whose address is in the calling sequence from the double-precision number in the A and B registers.

Fixed-point, double-precision multiplication (XDMU) multiplies the double-precision number whose address is in the calling sequence by the double-precision number in the A and B registers. XDMU uses double-precision addition of partial products.

Fixed-point, double-precision division (XDDI) divides the double-precision number in the A and B registers by the double-precision number whose address is in the calling sequence. XDDI returns the difference to the A and B registers.

Fixed-point, double-precision two's complement (XDCO) takes the two's complement of the double-precision number in the A and B registers. XDCO complements the number, then tests the low-order bits for a carry.

Fixed-point, integer absolute value (IABS) takes the absolute value of the signed integer in the A register. If the number is negative, IABS one's complements it, then corrects it to two's complement form.

Fixed-point, integer sign transfer (ISIG) applies the sign of the integer whose address is in the calling sequence to the quantity in the A and B registers.

Floating-Point Arithmetic

The floating-point subroutines provide higher accuracy, more flexibility, and wider number ranges than fixed-point arithmetic. Floating-point subroutines include:

a. Addition, subtraction, multiplication, and division

b. Absolute value

c. Sign copy

d. Mantissa separation

e. Normalization

Floating-point addition ($QK) algebraically adds the floating-point number in the A and B registers to the floating-point number whose address is in the calling sequence.

Floating-point subtraction (**$QL**) computes the difference of the floating-point minuend in the A and B registers and the floating-point subtrahend whose address is in the calling sequence.

Floating-point multiplication (**$QM**) multiplies the floating-point number in the A and B registers by the number whose address is in the calling sequence. $QM separates the mantissa and calls XDMU to implement the arithmetic operation.

Floating-point division (**$QN**) divides the floating-point number in the A and B registers by the number whose address is in the calling sequence. $QN separates the mantissa and calls XDDI to implement the arithmetic operation.

Floating-point, real-number absolute value (**ABS**) takes the absolute value of the floating-point, real quantity in the A and B registers. If the number is negative, ABS one's complements it and returns the result in the A and B registers.

Sign copy (**SIGN**) sets the sign of the floating-point number in the A and B registers equal to the sign of the quantity whose address is in the calling sequence.

The mantissa separation subroutines (**$FMS, $FSM**) separate the floating-point number in the A and B registers and return the mantissa in the A and B registers and the characteristic in the X register.

Normalization (**$NML**) normalizes the floating-point, double-precision number in the A and B registers. $NML tests the sign, two's complements the number using XDCO, and returns the fixed-point result in the A and B registers and the sign flag in the X register.

Arithmetic Functions

Subroutines are provided for the following arithmetic functions:

a. Logarithm

b. Exponential function

c. Square root

d. Sine

e. Cosine

f. Arctangent

g. Polynomial

h. Exponentiation

Fixed-point, single-precision logarithm (XLOG) computes the natural logarithm of the quantity in the A register. XLOG uses a Chebychev polynomial of the fifth degree.

Floating-point, double-precision logarithm (ALOG) computes the natural logarithm of the quantity whose address is in the calling sequence, returning the result in the A and B registers.

Fixed-point, single-precision exponential function, positive argument (XEXP) computes the exponential of the absolute value in the A register. It computes e^x divided by 4, where x is a positive fraction (between 0 and 1).

Fixed-point, single-precision exponential function, negative argument (XEXN) computes the exponential of the absolute value in the A register. It computes e^{-x}, where x is greater than zero and less than or equal to one.

Floating-point exponential function (EXP) computes the exponential of the floating-point quantity whose address is in the calling sequence.

Fixed-point, single-precision square root (XSQT) takes the unrounded square root of the quantity in the A register (if it is nonnegative) and returns the result in the A register.

Floating-point square root (SQRT) takes the square root of the floating-point number whose address is in the calling sequence.

Fixed-point, single-precision sine (XSIN) computes the sine of the quantity in the A register, returning the result in the A register.

Floating-point sine (SIN) computes the sine of the floating-point quantity whose address is in the calling sequence.

Fixed-point, single-precision cosine (XCOS) takes the cosine of the quantity in the A register and returns the result in the A register.

Floating-point cosine (COS) takes the cosine of the floating-point quantity whose address in the calling sequence.

Fixed-point, single-precision arctangent (XATN) computes the arctangent of the quantity in the A register, returning the result in the A register.

Floating-point arctangent (ATAN) computes the arctangent of the floating-point quantity whose address is in the calling sequence.

Fixed-point, single-precision polynomial (POLY) supports the fixed-point, single-precision mathematical subroutines that require the evaluation of a polynomial in one variable of any finite degree. The polynomial is evaluated in Horner form.

Fixed-point, integer exponentiation ($HE).

Integer/floating-point exponentiation ($PE).

Floating-point exponentiation ($QE).

Conversions

The number and character conversion subroutines include:

a. Fixed-point/floating-point

b. Binary/decimal

c. EBCDIC/Hollerith

d. EBCDIC/ASCII

e. Packed BCD/ASCII

Fixed-point, single-precision integer to floating-point conversion ($QS) converts the signed integer in the A register to floating-point format.

Floating-point to fixed-point, single-precision integer conversion ($HS) converts the floating-point number in the A and B registers to integer format.

Fixed-point, single-precision binary-to-decimal conversion (XBTD) converts the absolute value of the integer in the A register to a four-digit decimal-coded integer in the B register.

Fixed-point, single-precision decimal-to-binary conversion (XDTB) converts the four-digit, binary-coded-decimal integer in the A register to a pure binary integer in the B register.

EBCDIC-to-Hollerith conversion (SA01) converts and eight-bit EBCDIC character in the A register to its equivalent 12-bit Hollerith code, returning the result in the A register.

Hollerith-to-EBCDIC conversion (SB01) converts a 12-bit Hollerith code in the A register to its equivalent eight-bit EBCDIC character, returning the result in the A register.

EBCDIC-to-ASCII conversion (SC01) converts an eight-bit EBCDIC character in the A register to its equivalent eight-bit ASCII code, returning the result in the A register. This subroutine can be modified to produce seven-bit ASCII codes.

BCD-to-ASCII conversion (MT2A, n, s, e) converts a packed BCD character string of length n and beginning in location s, into a packed ASCII character string of length n beginning in location e.

ASCII-to-BCD conversion (A2MT, n, s, e) converts a packed ASCII character string of length n and beginning in location s into a packed BCD character string of length n beginning in location e.

Execution Times

Execution times for various mathematical subroutines are contained in the following three tables.

Double Precision Floating Point (45-bit mantissa, 8-bit exponent)

Operation	Execution Time (in milliseconds)		
	620/f-100	620/L-100	620/L
ADD	0.615	0.768	1.477
SUB	0.618	0.762	1.486
MUL	20.8	26.0	50
DIV	22.5	28.2	54
SQRT	295	369	644
SIN	109	136	261
COS	109	136	261
LOG	461	564	1130
EXP	334	417	802
ATAN	371	463	888

*Double precision math library does not utilize hardware multiply/divide.

Hardware Multiply/Divide

Operation	Execution Time (in microseconds)		
	620/f-100	620/L-100	620/L
ADD	1.5	2.0	3.6
SUB	1.5	2.0	3.6
MUL	6.38	10.0	18.0
DIV	6.38	10.0 to 14.0	18.0 to 25.0

Single Precision Floating Point (22-bit mantisa, 8-bit exponent)

Operation	Execution Time (in milliseconds)		
	620/f-100	620/L-100	620/L
ADD	0.140	0.175	0.337
SUB	0.171	0.214	0.410
MUL	0.236	0.295	0.566
DIV	0.362	0.452	0.869
SQRT	2.17	2.71	5.2
SIN	1.6	2.0	3.85
COS	1.6	2.0	3.85
LOG	1.76	2.2	4.23
EXP	1.36	1.7	3.3
ATAN	0.58 (min.) 3.25 (max.)	0.72 (min.) 4.06 (max.)	1.4 (min.) 7.8 (max.)

FORTRAN IV

FORTRAN IV

TABLE OF CONTENTS

SECTION 4
EXPRESSIONS AND ASSIGNMENTS

SECTION 5
CONTROL STATEMENTS

SECTION 6
INPUT/OUTPUT STATEMENTS

SECTION 7
PROGRAMS AND SUBPROGRAMS

SECTION 8
STAND-ALONE OPERATING PROCEDURE

SECTION 9
MOS AND VORTEX OPERATING PROCEDURES

SECTION 1 INTRODUCTION

Varian **FORTRAN IV** is a programming system for the Varian 73 and 620 computers and is comprised of a language, a library of subprograms, a compiler, and a run-time package (program). **FORTRAN IV** can be compiled and run as a stand-alone program, under the Master Operating System (MOS), or under the Varian Omnitask Real-Time Executive (VORTEX).

The **FORTRAN IV** language is especially useful in writing programs for scientific and engineering applications that involve mathematical computations. In fact, the name of the language **FORTRAN** is derived from its primary use: **FOR**mula **TRAN**slating. Source programs written in the **FORTRAN** language consist of a set of statements constructed from the elements described in this publication. The **FORTRAN** compiler analyzes the source program statements and transforms them into an object program that is suitable for execution. In addition, when the **FORTRAN** compiler detects errors in the source program, appropriate error messages are produced. The Varian **FORTRAN IV** language is compatible with and encompasses the American National Standards Institute (ANSI) **FORTRAN** (X3.9, 1966) including its mathematical subroutine provisions (except for the vertical spacing character + which is not implemented and is described in section 7.1.3.4 of the ANSI specification). Any valid programs compiled and executed using basic **FORTRAN** subset may also be compiled and executed by the **FORTRAN IV** compiler. Equivalent results are ensured by:

a. Common data formats.

b. Common format code routines.

c. Common mathematical subroutines.

d. Common libraries.

The following are salient features of the Varian 73/620 **FORTRAN IV**:

a. Scale Factor: The scale factor allows modification of data during conversion between internal and external representation.

b. Variable Attribute Control: The attributes of variables and arrays can be explicitly specified by statements or directives that:
   1. Specify the number of words assigned to an item.
   2. Explicitly type a variable as integer, real, double precision, complex, or logical.
   3. Specify the dimension of arrays.
   4. Specify data initialization values for variables.

c. Implied DO loops on DATA statements. An array can be easily preset to specific values.

d. Adjustable Array Dimensions: The dimensions of an array in procedure subprograms can be specified as variables. When the subprogram is called, the absolute array dimensions are substituted.

e. Three Dimension Arrays: An array has one, two, or three dimensions.

f. Six Character Variable Names: The name of a variable contains up to six characters.

g. Function subprograms return results via the argument list.

The first and largest part of this manual is devoted to the constructs of the **FORTRAN** language as implemented on Varian systems. This discussion logically proceeds from the most basic language elements to the general **FORTRAN** program structures.

The first subject presented is the characters, literals (constants and strings), and variables, the most basic units in **FORTRAN**. From these the programmer forms expressions and statements directing computation, storage and program control and the constructed related to input and output.

CHARACTER SET

A **FORTRAN** program unit is written using the following letters, digits, and special characters:

Letters: 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 $

Digits: 0 1 2 3 4 5 6 7 8 9

Special Characters:

	blank or space
=	equals
+	plus
–	minus
*	asterisk
/	slash
(	left parenthesis
)	right parenthesis
,	comma
.	decimal point

With the exception of the specific uses indicated in the following sections of this manual, a blank character has no meaning, and can be used freely by the programmer to improve the readability of the **FORTRAN** program.

The following special characters are classified as arithmetic operators and are significant in the unambiguous statement of arithmetic expressions:

+	addition or positive value
-	subtraction or negative value
*	multiplication
/	division
**	exponentiation

The special characters equals (=), open parenthesis (( ), close parenthesis ( )), comma (,), and decimal point (.), have specific application in the syntactical expression of the **FORTRAN** statement. The following sections of this manual qualify their use in particular statements and expressions.

In addition to the **FORTRAN** character set, the Varian 73/620 **FORTRAN IV** system accepts the following characters in Hollerith fields:

"	quotation mark	\	back slash
↑	uparrow	[	left bracket
!	exclamation	]	right bracket
#	number sign	<	less than
%	percent	>	greater than
&	ampersand	?	question mark
'	apostrophe	:	colon
;	semicolon		

Any other characters selected from the ASCII character set can also be accepted by the Varian 73/620 **FORTRAN IV**.

LINE FORMAT

A **FORTRAN** program consists of a series of statements divided into physical sections called lines that must be coded to a precise grammatical format. **FORTRAN** statements fall into two broad classes, executable and nonexecutable. Executable statements specify program action; nonexecutable statements describe the use of the program, the characteristics of the operands, editing information, statement functions, or data arrangement. The statements of a **FORTRAN** source program are normally written on a standard **FORTRAN** coding form.

Figure 1-1 is a sample **FORTRAN** coding form. The coding form includes 80 columns of information. Columns 73 through 80 are reserved for sequencing information, and have no effect upon the generated object program. Columns 1 through 72 contain program information in the format described below.

FORTRAN Coding Form

NAME | DATE | PROGRAM MATRIX MULTIPLICATION | PAGE

FORTRAN STATEMENT

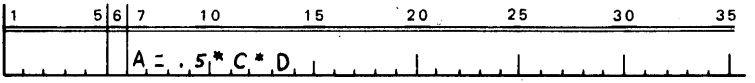
```
C     THIS PROGRAM READS IN THE VALUES FOR TWO MATRICES,
C     FORMS THEIR PRODUCT AND PRINTS THE RESULTS.
      DIMENSION A(5,20), B(20,10), C(5,10)
      READ (2,13) ((A(I,J), I=1,5), J=1,20),
     1            ((B(I,J), I=1,20), J=1,10)
      DO 2 I=1,5
      DO 2 J=1,10
      C(I,J)=0.0
      DO 2 K=1,20
    2 C(I,J)=C(I,J)+A(I,K)*B(K,J)
      WRITE (5,6) ((C(I,J), I=1,5), J=1,10)
   13 FORMAT (7E12.6)
    6 FORMAT (23H THESE ARE THE ANSWERS / (5E12.6))
      STOP 22
      END
```

Figure 1-1. Sample FORTRAN Coding Form

Initial Line

The first line of each statement is called an initial line. A statement line consists of three fields: statement number field, continuation flag, and statement field. A statement can include an initial line and continuation lines. Statements can have any number of continuation lines as required subject to the following restrictions: DO statements must have the first comma contained on an initial line; and the equals character (=) of a replacement statement or a statement function definition must appear on the initial line. An initial line can contain a statement label in columns 1 through 5. In this case, column 6 must contain a zero digit, blank, or space character; and columns 7 through 72 may contain all or part of a statement except for the restrictions noted.

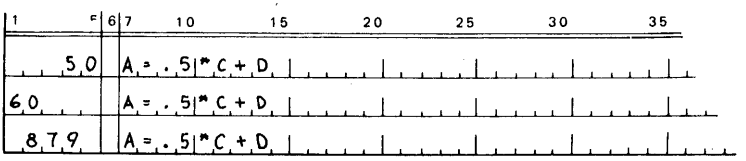
Example



Statement Number

Number permit statements to be referenced by other portions of a program. A statement number is an integer value in the range 1 to 99999 (leading zeros or blanks are not significant). The initial line of each statement may be given a unique number in columns 1 through 5. The same number cannot be given to more than one statement in a program unit.

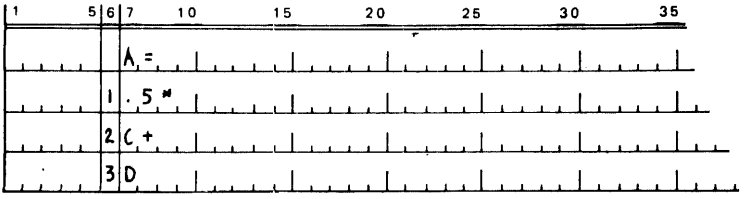
Example



Continuation Line

Continuation lines are used when additional lines of coding are required to complete a statement originating on an initial line. There can be any number of continuation lines per statement with the exceptions previously noted for initial lines. In a continuation line, columns 1 through 5 are blank. Column 6 contains any character other than a zero, blank, or space. The continuation of the statement is in columns 7 through 72.

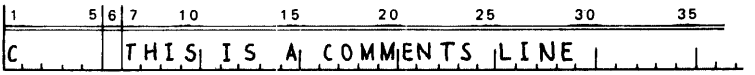
Example



Comment Line

Any line with the character C or an asterisk (*) in column 1 is identified as a comment line. Comments can appear anywhere in a program. All comment lines are ignored by a **FORTRAN** compiler, except for display purposes. Comments are in columns 2 through 72.

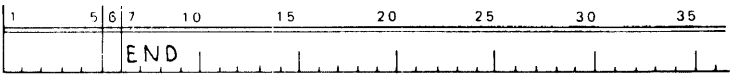
Example



End Line

Any line containing the character blank in columns 1 through 6 and having only the character string **END** in columns 7 through 72, preceded by, interspersed with, or followed by blank characters, is recognized by the processor as an end line to inform the processor that it has reached the physical end of the program.

Example



SECTION 2 — BASIC ELEMENTS

Constants and variables are distinguished in **FORTRAN** to identify the nature and characteristics of the values encountered in program execution. A constant is a quantity whose value is explicitly stated. A variable is a numeric quantity referenced by name, rather than by its explicit appearance in a program statement. During the execution of a program, a variable can assume many different values.

DATA TYPES

The Varian 73/620 **FORTRAN IV** compiler recognizes the following types of data: integer, real, double-precision, complex, logical, and Hollerith. Integer data are precise representations of integral values. Real data are approximations of real numbers. Both integer and real data may assume positive, negative, or zero values as follows (zero is considered neither positive nor negative):

	Range in 16-Bit Computers	Range in 18-Bit Computers
Integer	± 32,767	± 131,071
Real	Approx. $10^{\pm 38}$	Approx. $10^{\pm 38}$

DATA NAMES

FORTRAN data (variables, arrays, and array elements) are identified by names made up of letter or digit strings of one to six characters, the first character of which is a letter. (The character $ is processed exactly like a letter, but it is reserved for Varian system names. To avoid conflict, therefore, it is advisable not to use the $ character in names.) Data so identified are implicitly specified as being of type integer or real by the first character, although this can be changed by an explicit specification using a *TYPE* statement. In the absence of such an explicit specification, names beginning with the letters I, J, K, L, M, and N denote integers and other names denote real values.

Example of implicit integer names are:

I 12A MZXF N5

Examples of implicit real-number names are:

A B2 F5M79 AAA

CONSTANTS

Constant data are identified explicitly by giving their actual values. Constants do not change in value during program execution. They are specified as integer, real, double precision, complex, Hollerith, or logical constants.

Integer Constants

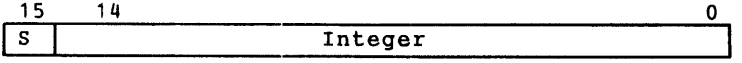
An integer constant is from one to five decimal digits without a decimal point. It can be preceded by a plus (+) or minus (−) sign. If the constant has no sign, it is interpreted as a positive value.

Example

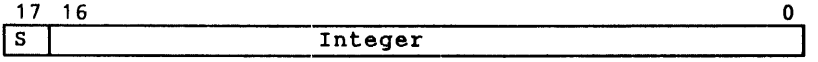
−217 −32767 +00327 512

In memory, an integer is stored in the format (two's complement):

16-Bit Computers



18-Bit Computers



Real Constants

A real constant may consist of from one to seven digits, a decimal point character, and an optional sign, plus or minus, or it may consist of a representation written in scientific notation. If a real constant is written in the latter form it must be formed from of one to seven decimal digits, an optional decimal point character, an optional sign character, followed by the letter E followed by one or two digits designating an exponent which may have an optional sign. In all case when a leading sign character is omitted, it is assumed to be positive. In FORTRAN notation the E portion of a real constant denotes that the value being represented is the number preceding the E multiplied by 10 raised to the power denoted by the integer constant following the E. The format of a real constant is:

± m.n

where, ± denotes an optional sign character, and m and n represent strings of decimal digits with a total combined characters not exceeding nine. Either m or n (but not both) may be omitted. An alternative form for a real constant, similar to scientific notation is:

± mpnE± d

where ± denotes an optional sign character, m and n represent strings of decimal digits, p is an optional decimal point which may be omitted only if n is omitted, and d is a one- or two-digit integer constant.

The following are equivalent real constants:

Examples

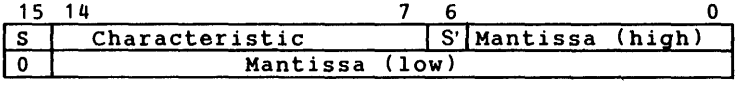
2E3	17.	-25.620E-1	0.0
2.E3	51E1	+.42	-.479
+2.E+03	-479E-3	.35E02	

The following are **invalid** real constants:

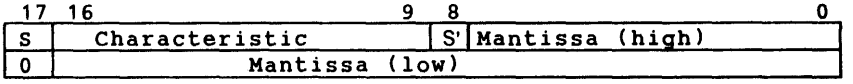
-1234	No decimal point or E part; interpreted as an integer literal
6.2E+99	Exceeds maximum size limit
6.2E-99	Smaller than minimum
9.8E072	Three-digit exponent part
E5	Exponent part alone not allow; taken as a variable name
1.2E3.4	Exponent part must be an integer
3E4E5	More than one exponent part
5,432.1	No commas or other punctuation allowed in real constant

In memory, a real number is stored in the format:

16-Bit Computers



18-Bit Computers



The characteristic is eight bits long with a bias of 0200. If the mantissa is negative, the entire first word is one's complemented. To represent zero, both words are set to zero. The bit S' indicates that the mantissa is normalized.

If a real constant is specified with more significant digits than the precision real data allow, truncation occurs, and only the most significant digits within the range will be represented. For a 16-bit computer, 6+ significant decimal digits are represented for a single-precision real constant and 13+ significant decimal digits for a double-precision real constant. For an 18-bit computer, the corresponding precisions are 7+ and 15+ significant decimal digits.

Double-Precision Constants

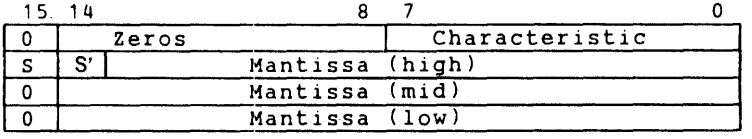
A double-precision constant in a source statement is specified exactly as an E-format real constant except that the letter E is replaced by D.

Example

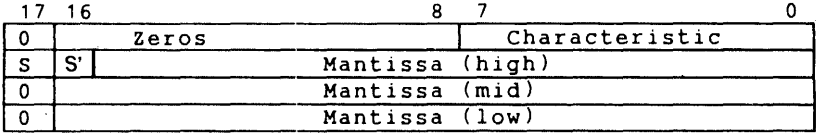
-3476.2D-4 28.D0 .578D + 3

In memory, a double-precision number is stored in the format:

16-Bit Computers



18-Bit Computers



The characteristic is eight bits long with a bias of 0200. If the mantissa is negative, the high-order word of the mantissa is in one's complement form. All four words are zero for the number zero. The bit S' indicates that the mantissa is normalized.

Complex Constants

A complex constant is formed by an ordered pair of signed or unsigned real single-precision constants separated by a comma and enclosed in parentheses.

The real constants in a complex constant can be positive, zero, or negative (if unsigned, they are assumed to be positive). The first real constant in a complex constant represents the real part of the complex number; the second represents the imaginary part of the complex number.

Examples

Valid Complex Constants

(−5.0E + 03,.16E + 02)	has the value −5000. + 16.01
(4.0E + 03,.16E + 02)	has the value 4000. + 16.01
(4.0E + 03,.16E + 02)	has the value 4000. + 16.0i
(2.1,0.0)	has the value 2.1 + 0.0i

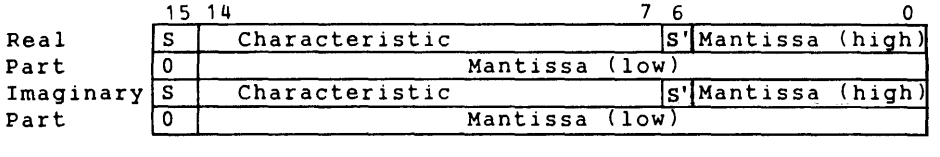
where i equals the square root of −1.

Invalid Complex Constants

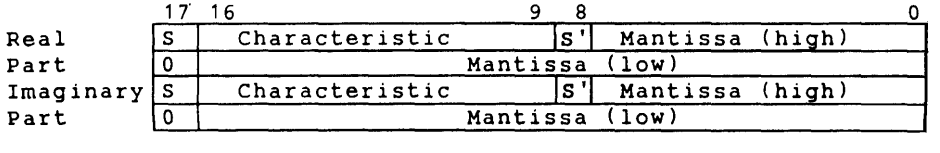
(292704,1.697)	the real part does not contain a decimal point
(1.2E113.279.3)	the real part contains an invalid decimal exponent
(.003D4,.005D6)	double-precision constants are invalid

In memory, a complex number is stored in the format:

16-Bit Computers



18-Bit Computers



Hollerith Constants

The general format of a Hollerith constant is:

```
n Hs
```

Where n is a positive non-zero constant denoting the number of characters in the string s which contains legal characters (see input formats) including blank.

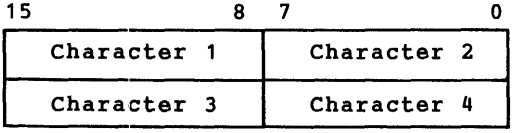
Any blank characters within the string will be considered part of the string, and should be counted. This is the only case in which embedded blanks are not ignored.

Examples

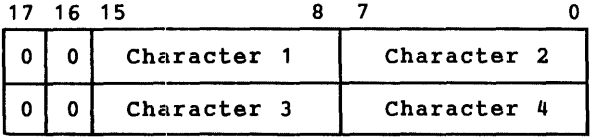
4HABCD
9HTEST CASE

In memory, a Hollerith constant is stored in the format:

16-Bit Computers



18-Bit Computers



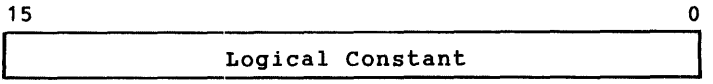
For Hollerith constants containing an odd number of characters, the last word contains the last character of the constant in the left byte and a blank in the right byte as character 2.

Logical Constants

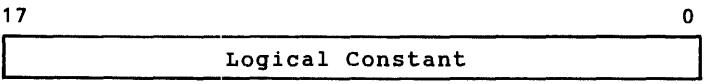
A logical constant specifies the logical value of a variable. There are two logical values: .TRUE. and .FALSE.. Each must be preceded and followed by a period as shown. The logical constants .TRUE. and .FALSE. specify that the value of the logical variable with which they are associated is true or false, respectively.

In memory, a logical constant is stored in the format:

16-Bit Computers



18-Bit Computers



where .FALSE. is stored as zero and .TRUE. is stored as minus one.

VARIABLES

A **FORTRAN** variable name is an identifier that consists of a string of one to six alphanumeric characters (letters and digits) with the leading character being a letter (including $). If the variable name is more than six characters long, then only the first six characters are retained internally. Embedded blanks are permitted within variable names but will be removed by the system.

Variables are classified into five basic types:
integer, real, double precision, complex, and logical.

A value represented by each of these types may be expressed by a literal of the same type. For instance, the value represented by an integer variable may be expressed by an integer constant. The value represented by each variable type must therefore comform to the same standards governing that type of constant.

Implicit Types

Unless declared otherwise in an explicit type statement, the variable name is assigned one of two types according to its initial letter. If the initial letter is I, J, K, L, M or N, the variable is an implicit integer type. If the first character is any other letter (or a dollar sign), the variable is an implicit real type.

The following are valid names of variables with their type implicitly assigned, (not overridden by an explicit assignment is a TYPE statement):

Name	Type
ABC123	Real
A$1	Real
INTEGER	Integer truncated to INTEGE 6 characters
I$A23	Integer
NO SUM	Integer, interpreted as NOSUM

The following are examples of **invalid** names of variables.

34SUM	Cannot begin with a numberic character
DATA-5	Cannot contain a character other that letters, numbers or a dollar sign.
$6.98	Period cannot be embedded in name

Note that double precision, complex and logical types are not assigned implicitly.

Arrays

FORTRAN variables can be grouped into two classes: simple variables that are identified by a single name formed by the definition above, and array elements that are designated by an array name followed by a subscript list enclosed in parentheses. An array is a convenient way to reference variables.

An array is an ordered set such that each member or element can be referenced by the array and subscripts can be used to denote the location in the dimensions.

The array name is formed by the same definition as a simple variable name.

The proper format of an array element is:

```
v (s)
```

Where v is a variable name and s is a subscript list which is a series of integer constants, variables, and arithmetic expressions.

A variable names is an array name only if it appears in an appropriate specification statement, such as a DIMENSION, TYPE or COMMON statement as a declarator. The declarator is used to set the maximum number and size of the dimensions allowed and must precede the first appearance of the array name in an executable or DATA statement. In a program, an identifier can be used as a simple variable name or an array name, but not both.

Whenever an array name appears in a FORTRAN program, the name must be immediately followed by a subscript list, except when it appears in:

a. A COMMON, DATA or type statement

b. The list of an I/O statement

c. The dummy argument list of a subprogram

d. The actual argument list of a subprogram reference

Each element of an array may be referenced by means of appropriate subscripts. Each entry in a subscript list is evaluated to obtain an integer value. Normally, the minimum value that the subscript of an array element can have is one. The maximum is the value specified in the array declarator.

SECTION 3 — SPECIFICATION STATEMENTS

Every executable **FORTRAN** program consists of a sequence of specification statements. These statements may be classified into executable and non-executable statements.

An executable statement causes an action at that point in the program when the program is executed.

A non-executable statement supplies information to the compiler when it is processing the **FORTRAN** statements. In general, these statements specify variable types, initial values, storage allocation, and allow subprograms to be used as actual arguments.

Specification statements organize and classify data that will be referred to by other statements in the **FORTRAN** program. Specification statements include:

DIMENSION	Names and declares the size of an array.
COMMON	Assigns variable and/or named arrays to common storage areas.
EQUIVALENCE	Assigns variables and named array to shared storage areas.
TYPE	Declares entities to be of type integer, real, double precision, complex, or logical.

Specification statements must precede all other statements except *TITLE, BLOCK DATA, FUNCTION, SUBROUTINE, EXTERNAL* and *NAME.*

DIMENSION STATEMENT

Form: DIMENSION V1(i1), V2(i2),..., Vn(in), where each V(i), (called an array declarator), is composed of a declarator name V (the name of the array), and a declarator subscript (i). Each (i) is an unsigned integer constant, two unsigned integer constants separated by a comma, or three unsigned integer constants separated by commas. Each constant must have a value greater than zero.

A *DIMENSION* statement specifies that the declarator names listed are arrays in the program unit. The number of dimensions and the maximum size of each dimension is specified by the declarator subscript associated with each declarator name.

More than one *DIMENSION* statement can appear in a program.

An array element is referred to by the array name qualified by a subscript to identify the desired element. If the value of this subscript is out of the range specified by the array declarator, the derived computational results will be unpredictable.

Array elements are stored column-wise in computer memory from low to high address storage. Therefore, one-dimension arrays are stored sequentially in the order A(1), A(2),....., A(n), while two-dimension arrays are stored with the first (leftmost) dimension varying most rapidly, i.e, A(1,1), A(2,1),....., A(m,1), A(1,2), A(2,2),....., A(m,n).

Example

```
DIMENSION A(5), I1(3,6), C(5,10), BIG(10,10,10)
```

This specification statement indicates that A is a real vector with five elements; I1 is an integer matrix of size 3 X 6 = 18 elements; C is a real matrix of size 5 X 10 = 50 elements; and BIG is a real matrix of size 10 X 10 X 10 = 1000 elements.

COMMON STATEMENT

General Form

```
COMMON /x/a,b,.../r/c,d,...
```

where:

a,b,...,c,d,... are variable or array names or array declarators.

/x/.../r/ represents optional common block names consisting of one through six alphanumberic characters, the first of which is alphabetic. These names must always be embedded in slashes.

Although the *COMMON* statement may be used to provide dimension information, adjustable dimensions may never be used. In the stand-alone and MOS systems, though not in VORTEX, the minimum size of any common memory block is 4095 words.

Variables or arrays that appear in a calling program or subprogram may be made to share the same storage locations with variables or arrays in other subprograms by use of the *COMMON* statements. For example, if one program contains the statement:

```
COMMON TABLE
```

as its first *COMMON* statement, and a second program contains the statement:

```
COMMON TREE
```

as its first *COMMON* statement and the two programs are loaded together, the variable names TABLE and TREE refer to the same storage location.

If the main program contains the statement:

```
COMMON A, B, C
```

and a subprogram contains the statement:

```
COMMON X, Y, Z
```

then A shares the same storage location as X, B shares the same storage location as Y, and C shares the same storage location as Z.

Common entries appearing in *COMMON* statements are cumulative in the given order throughout the program; that is, they are cumulative in the sequence in which they appear in all *COMMON* statements. For example, consider the following two *COMMON* statements:

```
COMMON A, B, C
COMMON G, H
```

These two statements have the same effect as the single statement:

```
COMMON A, B, C, G, H
```

Redundant entries are not allowed. For example, the following statement is invalid:

```
COMMON A, B, C, A
```

Consider the following example:

Example

CALLING PROGRAM

```
      .
      .
      .
COMMON A, B, C, R(100)
INTEGER R
      .
      .
      .
CALL MAPMY (...)
```

SUBPROGRAM SUBROUTINE

```
MAPMY(...)
      .
      .
      .
COMMON X, Y, Z, S(100)
INTEGER S
      .
      .
      .
```

Explanation:

In the calling program, the statement COMMON A, B, C, R(100) would cause 206 storage locations (two locations per variable) to be reserved in the COMMON area.

The statement COMMON X, Y, Z, S(100) would then cause the variables X, Y, Z, and S(1)...S(100) to share the same storage spaces as A, B, C, and R(1)...R(100), respectively.

From the above example, it can be seen that COMMON statements serve an important function: namely, as a medium to implicitly transmit data between the calling program and the subprogram. That is, values for X, Y, Z, and S(1)...(100), because they occupy the same storage locations as A, B, C, and R(1)...R(100), do not have to be transmitted in the argument list of a *CALL* statement. Arguments passed through COMMON must follow the same rules of presentation with regard to type, etc., as arguments passed in a list. (See the section entitled SUBPROGRAMS.)

In the preceding example, the common storage area (common block) established is called a blank common area. That is, no name was explicitly given to that area of storage (the name COMMON is assigned internally to the blank common block and will appear on maps). The variables that appeared in the *COMMON* statements were assigned locations relative to the beginning of the blank common area. However, variables and arrays may be placed in separate common areas. Each of these separate areas (or blocks) is given a name consisting of one through six alphanumeric characters (the first of which is alphabetic); those blocks which have the same name occupy the same storage space.

Those variables that are to be placed in labeled (or named) common are preceded by a common block name enclosed in slashes. For example, the variables A, B, and C will be placed in the labeled common area, HOLD, by the following statement:

```
COMMON/HOLD/A, B, C
```

In a *COMMON* statement, blank common can be distinguished from labeled common by preceding the variables in blank common by two consecutive slashes or, if the variables appear at the beginning of the common statement, by omitting any block name. For example, in the following statement:

```
COMMON A, B, C/ITEMS/X, Y, Z//D, E, F
```

the variables A, B, C, D, E, and F will be placed in blank common in that order; the variables X, Y, and Z will be placed in the COMMON area labeled ITEMS.

Blank and labeled common entries appearing in *COMMON* statements are cumulative throughout the program. For example, consider the following two *COMMON* statements:

```
COMMON A, B, C/R/D, E/S/F
COMMON G, H/S/I, J/R/P//W
```

These two statements have the same effect as the single statement:

```
COMMON A, B, C, G, H, W/R/D, E, P/S/F, I, J
```

COMMON is allocated from low to high memory addresses within a common block.

EQUIVALENCE STATEMENT

Form: EQUIVALENCE (k1), (k2),..., (kn), where each (ki) is a list of two or more nondummy variables and/or array element names, separated by commas. Subscript expressions of array element names must be nonzero, unsigned integer constants. An element of a two or three dimension array can be referred to by using a single subscript, giving the element position within the array (section 2.6.3).

The effect of the *EQUIVALENCE* statement is to cause the same area of memory to be shared by two or more entities. Each element of the ki list is assigned the same (or a part of the same) storage area.

More than one *EQUIVALENCE* statement is permitted in a program.

Example

```
DIMENSION A(5), I1(3,3), B1(3)
COMMON B, B1, B2
EQUIVALENCE (X, A(2), Y), (B, C2, F5), (I1(5), B2)
```

The effect of an *EQUIVALENCE* statement upon COMMON assignments may be the lengthening of COMMON. This lengthening is permitted only if it increases COMMON in the same direction as additional COMMON elements would. Thus, in this example, the equivalence (B1(1), A(3)) would have been invalid. It is also invalid to equate two elements of the same array to each other. Within a given list (Ki), no more than one element can be defined in a COMMON statement.

TYPE STATEMENT

General Form:

```
TYPE a, b,...., z
```

where:

TYPE is *INTEGER, REAL, DOUBLE PRECISION, COMPLEX,* or *LOGICAL*

a, b, ..., z represent variable or array names, array declarators, or function names

The *TYPE* statements declare the type *INTEGER, REAL, DOUBLE PRECISION,* COMPLEX or *LOGICAL* of a particular variable or array by its name, rather than by its initial character. This differs from the other way of specifying the type of a variable or array (i.e., implicit type specification). *LOGICAL* and *INTEGER* types are internally indistinguishable.

Example 1

INTEGER ITEM, VALUE

Explanation:

This statement declares that the variables ITEM and VALUE are of type *INTEGER.*

Example 2

REAL ARRAY, HOLD, VALUE, ITEM (5, 5)

Explanation:

This statement declares that the variables ARRAY, HOLD, VALUE, and the array named ITEM are of type REAL. In addition, it declares the size of the array ITEM. The variables ARRAY, HOLD, and VALUE have two storage locations reserved for each; and the array named ITEM has 50 storage locations reserved (two for each variable in the array).

Example 3:

DOUBLE PRECISION C, D, E

Explanation:

This statement declares that the variables C, D, and E are of type *DOUBLE PRECISION.* Thus, C, D, and E each have four storage locations reserved, one for the exponent and three for the mantissa (section 2.5.2.1).

Example 4

COMPLEX C, D, E

Explanation:

This statement declares that the variables C, D, and E are of type *COMPLEX.* Thus C, D, and E each have four storage locations reserved (two for the real part, two for the imaginary part).

SECTION 4 – EXPRESSIONS AND ASSIGNMENTS

EXPRESSIONS

Expressions specify the procedure by which a data value is obtained. An expression is any valid constant, variable, function reference, or a combination of these separated by appropriate operators and parentheses.

Expressions can be divided into two types: arithmetic and logical. If the type of literal which can represent the resulting value is true or false, then the expression is logical. An expression which yields a numeric quantity is an arithmetic expression.

The operators that can be used by a **FORTRAN** expression are listed in the table below with a relative precedence assigned to each operator by the compiler (the lowest number has the highest precedence).

OPERATOR	RELATIVE PRECEDENCE	FUNCTION
**	1	exponentiation
unary –	2	change of sign
/	3	division
*	3	multiplication
–	4	subtraction
+	4	addition
.NE.	5	not equal to
.GE.	5	greater than or equal to
.GT.	5	greater than
.EQ.	5	equal to
.LE.	5	less than or equal to
.LT.	5	less than
.NOT.	6	logical negation
.AND.	7	logical conjunction
.OR.	8	logical disjunction

The occurrence of these operators indicates that an arithmetic, logical, or, relational action is to be performed.

The arithmetic elements are described by the following statements:

PRIMARY	An *ARITHMETIC EXPRESSION* enclosed in parentheses, a constant, a variable reference, an array element reference, or function reference.
FACTOR	A *FACTOR* is a *PRIMARY* or a construct of the form: *PRIMARY**PRIMARY*
TERM	A *TERM* is a *FACTOR* or one of the forms: *TERM/FACTOR* *TERM*TERM*
SIGNED TERM	A *TERM* immediately preceded by a + or − sign.
SIMPLE ARITHMETIC EXPRESSION	A *TERM* or two *SIMPLE ARITHMETIC EXPRESSIONS* separated by a + or − sign.
ARITHMETIC EXPRESSION	A *SIMPLE ARITHMETIC EXPRESSION* or a signed *TERM* or either of the preceding immediately followed by a + or − sign and a *SIMPLE ARITHMETIC EXPRESSION.*

A *PRIMARY* of any type may be exponentiated by an *INTEGER PRIMARY* and the resulting factor is of the same type as that of the element being exponentiated. A *REAL* or *DOUBLE-PRECISION PRIMARY* may be exponentiated by a *REAL* or *DOUBLE-PRECISION PRIMARY.* The resultant *FACTOR* is of type *REAL* if both *PRIMARIES* are *REAL*, and otherwise of type *DOUBLE PRECISION.* These are the only cases for which use of the exponentiation operator is defined. Valid combinations for exponentiation are:

Base	**Exponent**
REAL	*REAL, INTEGER* or *DOUBLE PRECISION*
INTEGER	*INTEGER* (*REAL* and *DOUBLE PRECISION* exponents are invalid)
DOUBLE PRECISION	*REAL, INTEGER* or *DOUBLE PRECISION*

By use of the arithmetic operators other than exponentiation, any admissible element may be combined with another admissible element of the same type, and the resultant element is of the same type.

Further, an admissible real element may be combined with an admissible double-precision or complex element; the resultant element is of type *DOUBLE PRECISION* or *COMPLEX*, respectively.

A part of an expression is evaluated only if it is necessary to establish the value of the expression. The rules for formation of expressions imply the binding strength of operators. The range of the subrtraction operator is the term that immediately succeeds it. The evaluation may proceed according to any valid formation sequence. Use of an array element name requires the evaluation of its subscript. The type of the expression in which a function reference or subscript appears does not affect, nor is it affected by, the evaluation of the actual arguments or subscript. An element whose value is not mathematically defined cannot be evaluated.

The following rules represent the derivation of all permissible expressions:

A variable, constant, or function standing alone is an expression.

```
A(1)
JOBNO
217
17.26
SQRT(A+B)
```

If E is an expression whose first character is not an operator, then +E and −E are expressions.

```
-A(1)
+JOBNO
-217
+17.26
-SQRT(A+B)
```

If E is an expression, then (E) is an expression meaning the quantity E taken as a unit.

```
(-A)
-(+JOBNO)
-(X+Y)
(A-SQRT(A+B))
```

If E is an expression whose first character is not an operator, and F is an expression, then: F+E, F-E, F*E, F/E and F**E are all expressions.

```
-(B(I,J)+SQRT(A+B(K,L)))
-(B(I+E,3*J+K)+A)
1.7E-2**(X+5.0)
```

The mode of expression is determined by the modes of its elements, which must be the same with the following exceptions:

a. A *REAL* quantity can appear in an *INTEGER* expression only as an argument of a function.
   I + LFUNC(B)

b. An *INTEGER* quantity can appear in a *REAL* expression only as an argument of a function, or as a subscript or an exponent.
   AFUNC(I + 2)
   A(I,J + 1)
   B**N

The order of evaluation of expressions is established by the use of parentheses in the statement. If parentheses are not indicated, the following conventions of mathematics apply.

The hierarchy of operations, in order of precedence is: exponentiation, followed by multiplication and division, followed by addition and subtraction.

Within the same hierarchy of operations, evaluation proceeds from left to right.

Examples

X + Y*Z	is interpreted as	X + (Y*Z)
W*X/Y*Z	is interpreted as	((W*X)/Y)*Z
B**2- 4*A*C	is interpreted as	(B**2)- ((4.*A)*C)
X- Y- Z	is interpreted as	(X- Y)- Z
X/Y/Z	is interpreted as	(X/Y)/Z

ARITHMETIC ASSIGNMENT STATEMENT

General Form

a = b

where:

a is any subscripted or nonsubscripted variable.

b is any arithmetic expression.

This **FORTRAN** statement closely resembles a conventional algebraic equation; however, the equal sign specifies replacement rather than equivalence. That is, the expression to the right of the equal sign is evaluated, and the resulting value replaces the current value of the variable to the left of the equal sign.

Rules for Assignment of b to a

If a is *TYPE*	and b is *TYPE*	the assignment rule is*
INTEGER	*INTEGER*	Assign
INTEGER	*REAL*	Fix and Assign
INTEGER	*DOUBLE PRECISION*	Fix and Assign
INTEGER	*COMPLEX*	P
REAL	*INTEGER*	Float and Assign
REAL	*REAL*	Assign
REAL	*DOUBLE PRECISION*	DP Evaluate and Real Assign
REAL	*COMPLEX*	P
DOUBLE PRECISION	*INTEGER*	DP Float and Assign
DOUBLE PRECISION	*REAL*	DP Evaluate and Assign
DOUBLE PRECISION	*DOUBLE PRECISION*	Assign
DOUBLE PRECISION	*COMPLEX*	P
COMPLEX	*INTEGER*	P
COMPLEX	*REAL*	P
COMPLEX	*DOUBLE PRECISION*	P
COMPLEX	*COMPLEX*	Assign

***Notes**

P =	prohibited combination
Assign =	transmit the resulting value without change
Real assign =	transmit as much precision of the most significant part of the resulting value as a *REAL* datum can contain
DP evaluate =	evaluate according to the most precise rules, then DP float
Fix =	truncate any fractional part and transform to *INTEGER*
Float =	transform to *REAL*
DP float =	transform to *DOUBLE PRECISION* retaining as much precision as a *DOUBLE PRECISION* datum can contain

Assume that the type of the following variables has been specified as:

Variable Names	Type
I, J, W	*INTEGER* variables
A, B, C, D	*REAL* variables
E	*COMPLEX* variable

Then the following examples illustrate valid arithmetic statements using constants, variables, and subscripted variables of different types:

Statement	Description
A = B	The value of A is replaced by the current value of B.
W = B	The value of B is truncated to an integer value, and this value replaces the value of W.
A = I	The value of I is converted to a real value, and this result replaces the value of A.
I = I + 1	The value of I is replaced by the value of I + 1.
A = C*D	The most significant part of the product of C and D replaces the value of A.
E = (1.0,2.0)	The value of the complex variable E is replaced by the complex constant (1.0, 2.0). Note that the statement E = (A,B) where A and B are real variables, is invalid.

LOGICAL ASSIGNMENT STATEMENT

General Form

a = b

where:

a is a subscripted or nonsubscripted variable.

b is any logical expression.

Variable Names	Type
G, H	*LOGICAL* variables

Examples of logical assignment statements are:

Statement	Description
G = .TRUE.	The value of G is replaced by the logical constant .TRUE..
H = .NOT.G	If G is .TRUE., the value of H is replaced by the logical constant .FALSE.. If G is .FALSE., the value of H is replaced by the logical constant .TRUE..
G = 3..GT.I	The value of I is converted to a real value; if the real constant 3. is greater than this result, the logical constant .TRUE. replaces value of G. If 3. is not greater than I, the logical constant .FALSE. replaces the value of G.

LOGICAL EXPRESSIONS

The simplest form of logical expression consists of a single logical constant, logical variable, logical subscripted variable, or logical function reference, the value of which is always a truth value (i.e., either .TRUE. or .FALSE.).

More complicated logical expressions may be formed by using logical and relational operators. These expressions may be in one of the three following forms:

a. Relational operators combined with arithmetic expressions whose mode is *INTEGER*, *REAL*, or *DOUBLE PRECISION*.

b. Logical operators combined with logical constants (.TRUE. and .FALSE.), logical variables, subscripted logical variables, or logical function references.

c. Logical operators combined with either or both forms of the logical expressions described in items a and b.

Item a is discussed in the following section, Relational Operators; items b and c are discussed in the section entitled Logical Operators.

Relational Expressions

A relational expression consists of two arithmetic expressions separated by a relational operator and has the value .TRUE. or .FALSE. as the relation is true or false. Both arithemtic expressions can be type *INTEGER* (or LOGICAL) or one may be type *REAL* or

DOUBLE PRECISION and the other type *REAL* or *DOUBLE PRECISION*. If a real and a double precision expression appear in a relational expression, the effect is the same as a similar relational expression. This similar expression contains a double precision zero as the right-hand arithmetic expression and the difference of the two original expressions (in their original order) as the left. The relational operator is unchanged.

The six relational operators, each of which must be preceded and followed by a period, are as follows:

Relational Operator	Definition
.GT.	Greater than ($>$)
.GE.	Greater than or equal to ($\geq$)
.LT.	Less than ($<$)
.LE.	Less than or equal to ($\leq$)
.EQ.	Equal to ($=$)
.NE.	Not equal to ($\neq$)

The relational operators express an arithmetic condition which can be either true or false. Only arithmetic expressions whose mode is *INTEGER*, *REAL*, or *DOUBLE PRECISION* can be combined by relational operators. For example, assuming the type of variable has been specified as follows:

Variable Names	Type
ROOT, E, Q	*REAL* variables
A, I, F	*INTEGER* variables
L	*LOGICAL* variable
C	*COMPLEX* variable

then, the following illustrates valid and invalid logical expressions using the relational operators.

Example

Valid Logical Expressions Using Relational Operators:

```
(ROOT*Q).GT.E

A.LT.I

E**2.7.EQ.(5.*ROOT+4.)

57.9.LE.(4.7+E)

.5.GE.9.*ROOT

E.EQ.27.3E+05
```

(continued)

Invalid Logical Expressions Using Relational Operators:

C.LT.ROOT	Complex quantities can never appear in logical expressions.
C.GE.(2.7,5.9E3)	Complex quantities can never appear in logical expressions.
L.EQ.(A + F)	Logical quantities can never be compared to real quantities by relational operators.
E**2.EQ97.1E9	Missing period immediately after the relational operator.
.GT.9	Missing arithmetic expression before the relational operator.

Logical Operators

The three logical operators, each of which must be preceded and followed by a period, are as follows: (A and B represent logical constants or variables, or expressions containing relational operators).

Logical Operator	Definition
.NOT.	.NOT.A - if A is .TRUE., then .NOT.A has the value .FALSE.; if A is .FALSE., then .NOT.A has the value .TRUE.
.AND.	A.AND.B - if A and B are both .TRUE., then A.AND.B has the value .TRUE.; if either A or B or both are .FALSE., then A.AND.B has the value .FALSE.
.OR.	A.OR.B - if either A or B - or both are .TRUE., then A.OR.B has the value .TRUE.; if both A and B are .FALSE., then A.OR.B has the value .FALSE.

Two logical operators may appear in sequence only if the second one is the logical operator .NOT..

Only those expressions which, when evaluated, have the value .TRUE. or .FALSE. may be combined with the logical operators to form logical expressions. For example, assume that the type of variable has been specified as follows:

Variable Names	Type
ROOT, E, Q	*REAL* variables
A, I, F	*INTEGER* variables
L, W	*LOGICAL* variables
C	*COMPLEX* variable

Then the following examples illustrate valid and invalid logical expressions using both logical and relational operators.

Examples

Valid Logical Expressions:

(ROOT*Q.GT.E).AND.W

L.AND..NOT.(I.GT.F)

(E+5.9E2.GT.2.*E).OR.L

.NOT.W.AND..NOT.L

L.AND..NOT.W.OR.I.GT.F

(E**F.GT.ROOT).AND..NOT.(I.EQ.A)

Invalid Logical Expressions

E.AND.L	E is not a logical expression.
.OR.W	.OR. must be preceded by a logical expression.
NOT.(A.GT.F)	missing period before the logical operator .NOT.
(C.EQ.I).AND.L	a complex variable may never appear in a logical expression.
L.AND..OR.W	the logical operators .AND. and .OR. must always be separated by a logical expression.
.AND.L	.AND. must be preceded by a logical expression.

Order of Computations in Logical Expressions

Where parentheses are omitted, or where the entire logical expression is enclosed within a single pair of parentheses, the order in which the operations are performed is as follows:

Operation	Hierarchy
Evaluation of Functions	1st (highest)
Exponentiation (**)	2nd
Multiplication and division (* and /)	3rd
Addition and subtraction (+ and -)	4th
.LT.,.LE.,.EQ.,.NE.,.GT.,.GE.	5th
.NOT.	6th
.AND.	7th
.OR.	8th

For example, the expression:

```
(A.GT.D**B.AND..NOT.L.OR.N)
```

is effectively evaluated in the following order.

1. D**B	Call the result W (exponentiation)
2. A.GT.W	Call the result X (relational operator)
3. .NOT.L	Call the result Y (highest logical operator)
4. X.AND.Y	Call the result Z (second highest logical operator)
5. Z.OR.N	Final operation

Use of Parentheses in Logical Expressions

Parentheses may be used in logical expressions to specify the order in which the operations are to be performed. Where parentheses are used, the expression contained within the most deeply nested parentheses (that is, the innermost pair of parentheses) is effectively evaluated first. For example, the logical expression:

```
((I.GT.(B + C)).AND.L)
```

is effectively evaluated in the following order.

1. B + C	Call the result X
2. I.GT.X	Call the result Y
3. Y.AND.L	Final operation

The logical expression to which the logical operator .NOT. applies must be enclosed in parentheses if it contains two or more quantities. For example, assume that the values of

the logical variables, A and B, are .FALSE. and .TRUE., respectively. Then the following two expressions are not equivalent:

```
.NOT.(A.OR.B)
.NOT.A.OR.B
```

In the first expression, A.OR.B is evaluated first. The result is .TRUE., but .NOT. (.TRUE.) implies .FALSE.. Therefore, the value of the first expression is .FALSE..

In the second expression, .NOT.A is evaluated first. The result is .TRUE.; but .TRUE.OR.B implies .TRUE.. Therefore, the value of the second expression is .TRUE..

SECTION 5 — CONTROL STATEMENTS

Each statement in a **FORTRAN** program is executed in the order of its appearance in the source program, unless this sequence is interrupted or modified by a control statement. The control statements are: *GO TO, IF, CALL, RETURN, CONTINUE, PAUSE, STOP,* and *DO.*

GO TO STATEMENTS

GO TO statements transfer logical control from one section of a program to another. **FORTRAN** includes three forms of the *GO TO* statement: unconditional, computed, and assigned *GO TO.*

Unconditional GO TO

An unconditional *GO TO* is of the form: GO TO k, where k is a statement label reference.

Execution of this statement causes the statement identified by the label k to be executed next in sequence.

Example

```
      GO TO 72
        .
71  V7 = HQ(5) + Y**L
        .
        .
        .
72  V7 = HQ(4) + X**J
```

In this example, execution of the GO TO 72 statement causes statement number 71 and any succeeding statements to be bypassed. Execution is resumed with statement number 72.

Computed GO TO

The computed *GO TO* statement is of the form: GO TO (k1, k2, ...,kn), i, where the k's are statement label references, and i is an integer variable reference.

Execution of this statement causes the statement identified by the statement label kj to be executed next in sequence, where j is the value of i at execution time. Valid execution of

this statement is dependent upon the value of the integer variable such that 1 is less than or equal to j, and j is less than or equal to n.

Example

```
GO TO (98, 405, 3), n
```

Execution of the statement in the example will cause control to be transferred to the statement labeled 98, 405, or 3 if the value of the variable integer n is 1, 2, or 3, respectively.

ASSIGN and Assigned GO TO

General Form

```
ASSIGN i TO m
      .
      .
      .
GO TO m, (X1, X2, X3,...,Xn)
```

where:

i is an executable statement number.

X1,X2,X3,...,Xn are executable statement numbers.

m is a nonsubscripted integer variable that is assigned one of the following statement numbers: X1, X2, X3,...Xn.

The assigned *GO TO* statement causes control to be transferred to the statement numbered X1, X2, X3,..., or Xn, depending on whether the current assignment of m is X1, X2, X3,..., or Xn, respectively. For example, in the following statement:

```
GO TO N, (10, 25, 8)
```

If the current assignment of the integer variable N is statement number 8, then the statement numbered 8 is executed next. If the current assignment of N is statement number 10, the statement numbered 10 is executed next. If N is assigned statement number 25, statement 25 is executed next.

The current assignment of the integer variable m is determined by the last executed *ASSIGN* statement. Only an *ASSIGN* statement may be used to initialize or change the value of m. The value of m is not the integer statement number; ASSIGN 10 TO I is not the same as I = 10.

Example 1

```
      .
      .
      .
   ASSIGN 50 TO NUMBER
      .
10 GO TO NUMBER, (35, 50, 25, 12, 18)
      .
      .
      .
50 A=B+C
      .
      .
      .
```

In the above example, statement 50 is executed immediately after statement 10.

Example 2

```
      .
      .
      .
   ASSIGN 10 TO ITEM
      .
      .
      .
13 GO TO ITEM, (8, 12, 25, 50, 10)
      .
      .
      .
 8 A=B+C
      .
      .
      .
10 B=C+D
      .
   ASSIGN 25 TO ITEM
   GO TO 13
      .
      .
      .
25 C=E**2
      .
      .
      .
```

In the above example, the first time statement 13 is executed, control is transferred to statement 10. On the second execution of statement 13, control is transferred to statement 25.

ARITHMETIC IF STATEMENT

It is often necessary to alter the logical flow of a program on the basis of the results of an arithmetic test. The *IF* statement is a conditional transfer that will execute this level of control, and is of the form:

```
IF (e) k1, k2, k3
```

The arithmetic *IF* is a three-way transfer. Execution of this statement causes the expression (e) to be evaluated, following which the statement identified by the label k1, k2, k3 is executed next in sequence, as the value of (e) is less than zero, equal to zero, or greater than zero, respectively.

Example

```
    IF (I) 10, 11, 12
10  V7 = HQ(5) + Y**L

    GO TO 13
11  V7 = HQ(4) + X**J

    GO TO 13
12  V7 = HQ(3) + X**L

13  NEXT STATEMENT
```

In this example, execution of IF (I) 10, 11, 12 causes one of the following actions: for a negative value of I, statement number 10 is executed in sequence; for a zero value of I, statement number 10 and any succeeding statements are bypassed and statement number 11 is executed; for a positive nonzero value of I, statements 10 through 11 and any statement following statement 11 are bypassed, and statement number 12 is executed.

Logical IF Statement

General Form

```
IF (a)s
```

where:

a is any logical expression.

s is any executable statement except a *DO* statement or another logical *IF* statement

The logical *IF* statement is used to evaluate the logical expression (a) and to execute or skip statement s depending on whether the value of the expression is .TRUE. or .FALSE., respectively.

Example

```
      .
      .
      .
 5  IF (A.LE.0.0) GO TO 25
10  C=D+E
15  IF (A.EQ.B) ANSWER = 2.0*A/C
20  F=G/H
      .
      .
      .
25  W=X**Z
      .
      .
      .
```

In statement 5, if the value of the expression is .TRUE. (i.e., A is less than or equal to 0.0), the statement GO TO 25 is executed next and control is passed to the statement numbered 25. If the value of the expression is .FALSE. (i.e., A is greater than 0.0), the statement GO TO 25 is ignored and control is passed to the statement numbered 10.

In statement 15, if the value of the expression is .TRUE. (i.e., A is equal to B), the value of ANSWER is replaced by the value of the expression (2.0*A/C), and statement 20 is executed. If the value of the expression is .FALSE. (i.e., A is not equal to B), the value of ANSWER remains unchanged and statement 20 is executed next.

Example

```
      .
      .
      .
 5  IF (P.OR..NOT.Q) A=B
10  C=B**2
      .
      .
      .
```

Assume that P and Q are logical variables. In statement 5, if the value of the expression is .TRUE., the value of A is replaced by the value of B and statement 10 is executed next. If the value of the expression is .FALSE., statement A=B is skipped and statement 10 is executed.

CALL STATEMENT

The *CALL* statement causes a transfer of execution control to a subroutine-type subprogram, and is of one of the forms: CALL s(a1, a2,..., an) and CALL s, where s is the name of a subroutine and the a's are actual arguments that will replace the dummy arguments in the called subroutine. Arguments can be Hollerith constants, variable names, array element names, array names, any other expression, or the name of an external procedure. They must, however (except for Hollerith constants), be indicated in order, number, and type with the corresponding dummy arguments of the subroutine. External procedure names must be declared by an EXTERNAL statement.

Execution of the *CALL* statement transfers control to the designated subroutine. The arguments declared in the statement line are associated with the dummy arguments that are parameters of the executable statements of the subroutine. Control is then passed to the first executable statement of the called subroutine. Control will be returned to the first executable statement following the *CALL* statement upon execution of the *RETURN* statement in the subroutine. Examples of calling sequences to subroutines are shown below.

```
CALL TEST (A,I)
CALL EXIT
```

The first example will transfer execution control to the subroutine labelled *TEST* and include the parameters or arguments A and I in the subroutine. The second example will cause execution control to be transferred to the subroutine labelled *EXIT*. Any arguments required for execution of exit are self-contained in the logic of the subroutine.

RETURN STATEMENT

The execution of a *RETURN* statement results in the exit from a subprogram, and is expressed in the form: RETURN. A *RETURN* statement defines the logical end of a procedure subprogram and, therefore, may appear only in a subprogram. Execution of the statement returns logical control to the current calling program unit. Each subprogram must contain at least one *RETURN* statement.

In the case of a subroutine subprogram, control is returned to the first statement immediately following the *CALL* statement that released control to the subroutine. In the case of a function subprogram, control is returned (with the value of the function available) to the statement that called the function subprogram.

CONTINUE STATEMENT

Form: CONTINUE.

The *CONTINUE* statement results in no action in an execution sequence; therefore, the statement has no effect upon the program. This statement serves as a program unit reference point and is frequently used at the end of a *DO* loop.

Example

```
    IF (I) 10, 11, 12
10  V7 = HQ(5) + Y**L
      .
      .
    GO TO 13
11  V7 = HQ(4) + X**J
      .
      .
    GO TO 13
12  V7 = HQ(3) + X**L
      .
      .
13  CONTINUE
```

PAUSE STATEMENT

Form: PAUSE n or PAUSE, where n is octal digit string of from one to five digits designating the particular *PAUSE*.

A *PAUSE* statement causes a temporary cessation of program execution and displays PAUSE n on the console device (logical unit SO for MOS and VORTEX). The statement permits operator intervention for setup or control functions, such as changing data tapes. For stand-alone and MOS systems, the computer executes a halt instruction, delaying further execution until the computer is placed in the run mode. For VORTEX, a SUSPND call is executed that suspends program execution until a RESUME call is made (see VORTEX Reference Manual, 98 A 9952 101) for descriptions of SUSPEND and RESUME calls. Execution will resume at the first executable statement following the *PAUSE* statement.

Example

```
PAUSE 01
```

When executed under VORTEX, the task name precedes the *PAUSE* statement.

STOP STATEMENT

Form: STOP n or STOP, where n is an octal digit string of from one to five digits designating the particular *STOP*.

A *STOP* statement causes termination of program execution and displays STOP n (see section 8 for display format). The program then terminates.
MOS and VORTEX terminations occurs with exit calls. In the stand-alone system, terminations occur with the execution of a hardware halt.

Example

```
STOP 0721
```

When executed under VORTEX, the task name precedes the *STOP* statement.

DO STATEMENT

The *DO* statement controls repetitive execution of a group of statements. The number of repetitions depends on the value of a control variable. The statement assumes one of the forms: DO n i = m1, m2, m3 and DO n i = m1, m2, where n is the statement label of an executable statement. This statement, called the terminal statement of the associated *DO* must physically follow and be in the same program unit as the *DO* statement. The terminal statement may not be a *GO TO* of any form, arithmetic *IF*, *RETURN*, *STOP*, *PAUSE*, or another *DO* statement, nor a logical *IF* statement containing one of these forms.

Symbol i is an integer variable name, identified as the control variable.

Symbol m1, identified as the initial parameter, m2, as the terminal parameter, and m3, as the incrementation parameter are each either an integer constant or integer variable reference. If the second form of the *DO* statement is used, a value of 1 is implied for the incrementation parameter. When the *DO* statement is executed, the values of m1, m2, and m3 must be greater than zero.

Associated with each *DO* statement is a range that is defined to be those executable statements from and including the first executable statement following the *DO*, to and including the terminal statement defined by the *DO*. A special situation, called nesting, occurs when the range of a *DO* contains another *DO* statement. In this case, the range of the contained *DO* must be a subset of the range of the containing *DO*. There is no limit to the nesting of *DO* statements.

The control variable is assigned the value represented by the initial parameter. This value must be less than or equal to the value represented by the terminal parameter.

The range of the *DO* is executed.

If control reaches the terminal statement after execution of the terminal statement, the control variable of the most recently executed *DO* statement associated with the terminal statement is incremented by the value represented by the associated incrementation parameter.

If the value of the control variable is greater than the value represented by its associated terminal parameter, the *DO* is said to be satisfied, and the control variable becomes undefined.

If there were one or more other *DO* statements referring to the terminal statements in question, the control variable of the next most recently executed *DO* statement is incremented by the value represented by the associated incrementation parameter until all *DO* statements referring to the particular termination statement are satisfied, at which time the first executable statement following the terminal statement is executed.

Upon exiting from the range of a *DO* by execution of a *GO TO* statement or an arithmetic *IF* statement, that is other than by satisfying the *DO*, the control variable of the DO is defined and is equal to the most recent attained value.

A *GO TO* or arithmetic *IF* statement may not cause control to pass into the range of a *DO* from outside its range. When a procedure reference occurs in the range of a *DO*, the actions of that procedure are considered to be temporarily within that range, i.e., during the execution of that reference.

The control variable, initial, terminal, and incrementation parameters of a *DO* may not be redefined during the execution of the range of that *DO*.

If a statement is the terminal statement of more than one *DO* statement, the label of that terminal statement may not be used in any *GO TO* or arithmetic *IF* statement that occurs anywhere but in the range of the most deeply contained *DO* with that terminal statement.

Example

```
DO 607 K1 = 2, ID, 3
```

The foregoing statement would cause K1, the control variable, to be set to the value of the initial parameter, 2. Execution would proceed at the statement immediately following, down to and including the statement identified by the label 607. After each execution of the loop, K1 is incremented by the incrementation parameter, 3, and evaluated in relation to the current value of the terminal parameter, ID. If the current value of K1 = ID, execution control is transferred to the statement following that identified by the label 607; otherwise, the *DO* cycle is repeated.

Example illustrating *DO* nesting:

```
      WRITE (MX,8)
      L = 0
      DO 150 J = 1,K
      DO 140 I = 1,M
      L = L + 1
140   D(I) = V(L)
150   WRITE (MX,9)J,(D(I),I = 1,M)
      CALL LOAD (M,K,R,V)
C     PRINT FACTOR MATRIX
      WRITE (MX,10)K
      DO 180 I = 1,M
      DO 170 J = 1,K
      L = M*(J-1) + I
170   D(J) = V(L)
180   WRITE (MX,11)I,(D(J),J = 1,K)
      IF (K-1) 185, 185, 188
185   WRITE (MX,19)K
      GO TO 100
188   CALL VARMX (M,K,V,NC,TV,B,T,D)
```

SECTION 6 INPUT/OUTPUT STATEMENTS

Input statements provide a program with the means of receiving information from external sources. Output statements allow the transmission of program data to external sources. These external sources may be devices such as magnetic tape and paper tape handlers, typewriters, and punch card processors.

There are two types of input/output statements.

READ and *WRITE* statements
AUXILIARY Input/Output statements

The first statement type causes the transfer of records of sequential files to and from the program. These data may be formatted information consisting of strings of characters or unformatted information consisting of binary word values in the form in which they normally appear in storage. The second statement type consists of the *BACKSPACE* and *REWIND* statements, which provide for positioning of devices and the *ENDFILE* statement, which provides for writing an end of file indicator.

Input/output statements reference input/output units, formatted information, and format specifications. An input/output unit is identified by a **FORTRAN** unit number u that can be an integer constant or a variable name (or array element) referencing an integer constant. All input/output statements for stand-alone and MOS **FORTRAN** programs must contain explicit references to unit numbers at compiling time (e.g., REWIND 7, READ (2, 6)). Under VORTEX, input/output statements may contain implied references to unit numbers at compiling time (e.g., WRITE (J, 15), or REWIND M5).

The format specification f is defined by either a *FORMAT* statement having the statement label f, or an array name. If f is a *FORMAT* statement label, the statement must appear in the same program as the input/output statement.

INPUT/OUTPUT LISTS

The input list specifies the names of variables and array elements to which input values are assigned. The output list specifies the names of variables and array elements whose values are to be transmitted. Input and output lists are of the same form.

SIMPLE LISTS

Simple lists have the form: m1, m2, m3...., mn, where mi is the name of a variable or array element. Commas separate each name in the list. The period signifies possible additional list items. List elements can be enclosed in parentheses.

Example

Input Lists	Output Lists
A	B
C(26,L)	I(10,10)
R, K, D, (I, J)	S, (R, K), F(1,25)

An array variable in a list that is not subscripted is considered equivalent to the listing of each successive element of the array. If B is an array, list B is equivalent to B (1, 1), B (2, 1), B (3, 1),...., B (1, 2), B (2, 2),...., B (j, k), where j and k are the subscript limits of B.

DO-IMPLIED LISTS

A *DO*-implied list is a simple list followed by a comma character and an expression of the form: i = m1, m2, m3 or i = m1, m2.

The elements i, m1, m2, and m3 have the same meaning as defined for the *DO* statement. The *DO* implication applies to all simple list items enclosed in parentheses with the implication. For input lists, i, m1, m2, and m3 may appear within this range only as subscripts.

Examples

DO-Implied Lists:

```
(X (I), I = 1, 4)
(Q (J), R (J), J = 1, 2)
(G (K), K = 1, 7, 3)
((A (I, J), I = 3, 5), J = 1, 2)
(X (K), K = 1, 2), I, (R (J), J = 3, 5)
```

Equivalent Simple Lists:

```
X (1), X (2), X (3), X (4)
Q (1), R (1), Q (2), R (2)
G (1), G (4), G (7)
A (3, 1), A (4, 1), A (5, 1)
A (3, 2), A (4, 2), A (5, 2)
X (1), X (2), I, R (3), R (4), R (5)
```

READ STATEMENTS

These statements are used to obtain data values from an external source. The data values are input in either formatted or unformatted mode. The form of a formatted *READ* statement is:

```
READ (u,f) k.
```

The verb READ and the parentheses must appear in this form.

Execution of this statement causes information to be transmitted from the external source whose **FORTRAN** unit number is defined by u. These data are scanned and converted as specified by the format specification, f, and the resulting values are assigned to the variable names defined in the list, k.

The form of an unformatted *READ* statement is: READ (u) k.

The verb READ and the parentheses must appear in this form.

This statement causes data to be input in binary form from the unit defined by u. The values are assigned to the variable names defined in the list, k.

Examples

```
READ (1,44) A, B, C
READ (2) R, S
READ (N, 12) A, (R (I), I = 1, 10)
READ (L) S, (T (J), J = 1, N)
```

All information appearing on external sources is divided into records. Each time a *READ* statement is executed, a new record is processed. The number of records input by a single *READ* statement is determined by the list and format specification. If only part of a record is input the remainder of the record is lost as the next *READ* processes the next record. Records are read sequentially until the list is exhausted. Only enough values are read to fill the list.

The list, k, in an unformatted *READ* statement may be left blank to skip a record.

Formatted and unformatted records are described in sections 8 and 9.

WRITE STATEMENTS

WRITE statements are used to transfer program data to external devices. These data may be formatted or unformatted. The form of a formatted *WRITE* statement is: WRITE (u, f) k.

The verb WRITE and the parentheses must appear in this form.

Execution of this statement causes records to be written on the device referenced by u. The contents of the records are the values taken sequentially from the list, k, converted according to the format specification, f.

The form of an unformatted *WRITE* statement is:

```
WRITE (u) k.
```

The verb WRITE and the parentheses must appear in this form.

Execution of this statement causes binary information from the list, k, to be written in records on the unit defined by u.

Examples

```
WRITE (1, 4) A, B, C
WRITE (7) R, S, T
WRITE (K, 12) X, (Y (J), J = 1, M), I
WRITE (N) W, Z, (F (K), K = 1, 5)
```

Several records may be written with a single *WRITE* statement. The number of records is determined by the list and format specifications. Successive records are written until the data are exhausted. If the data do not fill a record, the record is filled with blanks.

REWIND STATEMENT

This statement is of the form:

```
REWIND u.
```

Execution of this statement causes the physical unit defined by u to be rewound.

BACKSPACE STATEMENT

This statement has the form:

```
BACKSPACE u.
```

The *BACKSPACE* statement causes the physical unit defined by u to be backspaced one record.

ENDFILE STATEMENT

This statement has the form:

```
ENDFILE u.
```

When this statement is executed, an end of file indicator is written on the physical unit defined by u.

FORMAT STATEMENTS

FORMAT statements, with input/output operations, specify conversion and editing of information between program storage and external representation. *FORMAT* statements

are nonexecutable and must have a statement label to be referenced by input/output statements. Conversion performed according to a *FORMAT* statement during output is in general the reverse of conversion performed during an input operation.

A *FORMAT* statement is expressed as:

n FORMAT (f1, f2, f3, ..., fn)

where

n is the statement label and the fn are field specifications

The noun *FORMAT* and the parentheses must appear in this form.

When formatted records are output to a printer, the first character of the record is not printed but is processed as a printer vertical spacing control character as follows:

Character	Vertical Spacing
blank	One line
0	Two lines
1	To first line of next page

The ANSI no-advance character + is not implemented for the MOS and stand-alone systems. Refer to the VORTEX Reference Manual for the devices that process the + character.

FIELD SPECIFICATIONS

FIELD specifications describe the type of conversion and editing to be performed on each variable appearing in the input/output list. *FIELD* specifications can be in any of the following forms:

rAw rFw.d rEw.d rDw.d rIw nHs 's' nX rLw rGw.d

where:

a. The characters A, D, E, F, G, L, and I indicate the manner of conversion for variables in the list.

b. The characters H, 's', and X represent characters to be input/output directly from the format ('s' is used with VORTEX only).

c. The character / represents the end of a record.

d. w and n are nonzero integer constants defining the width of the field (including digits, decimal points, and algebraic signs) in the external character string.

(continued)

e. d is an integer specifying the number of fractional digits appearing in the external string.

f. r is an optional, nonzero integer indicating that the specification is to be repeated r times.

g. s is a string of acceptable **FORTRAN** characters.

h. The T specification relocates the current position in the external record (VORTEX only).

i. Y is a non-zero integer constant specifying the character position in the external record.

F CONVERSION

General form:

rFw.d

Only real data may be processed by this form of conversion.

Output

The field is right-justified with as many leading blanks as necessary to fill w. Negative values are preceded by a minus sign. Internal values are converted to fixed-point decimal numbers and rounded to d decimal places.

For a field specification of F10.4:

368.4	is converted to	368.4000
12.0	is converted to	12.0000
-17.90767	is converted to	-17.9077
-37.5E-2	is converted to	.3750

If a value requires more positions than allowed by w, the most significant digits, including sign if negative, are output. The error indication is designated by an asterisk in the least significant character position.

For a field specification of F6.4:

4739.76	is converted to	4740*
-12.463	is converted to	-12.5*

Input

Input strings are decimal numbers of length w with d characters in the fractional portion. Blanks are treated as zeros. If a decimal point is present in a value, the fractional portion of the value is explicitly defined by that decimal point character.

For a field specification F8.3:

35	is converted to	0.035
964372	is converted to	964.372
0.53821	is converted to	0.53821
-16.402	is converted to	-16.402
-12	is converted to	-0.012
47.-4	is converted to	0.0047

E CONVERSION

General form:

rEw.d

Only real data may be processed by this form of conversion.

Output

Internal values are converted to decimal values of the forms:

.ddd...dE+ee and .ddd...E-ee

where:

ddd...d represents d digits, and
ee is a decimal exponent.

The leading decimal point and E characters are present exactly as shown. Internal values are rounded to d digits, and negative values are preceded by a minus sign. The external field is right-justified and preceded by blanks to fill the width, w. This field width includes the exponent digits, the sign of the exponent (minus or space), the letter E, the magnitude digits, the decimal point, and the sign of the value (minus or space). This means that the field width should correspond to the relation: $w \geq d + 6$.

If w is less than (d + 6), the format is in error.

For a field specification of E12.5:

76.573	is converted to	.76573E 02
58796.341	is converted to	.58796E 05
-369.7583	is converted to	-.36976E 03
0.006873	is converted to	.68730E-02
0.2	is converted to	.20000E 00
-0.0000054	is converted to	-.5400E-05

Input

Each external value is of field width w with d characters in the fractional part of the value. The value is right-justified with all blanks counting as zeros. A minus sign may precede the value of the exponent. A decimal point placed in the fractional part takes precedence over the d specification. The character E may be present to separate the value and the exponent.

For a field specification of E10.3:

123E3	is converted to	123.0
12874E2	is converted to	1287.4
-563E-02	is converted to	-0.00563
398E00	is converted to	0.398
5387601	is converted to	538.76
5455-01	is converted to	0.5455
-6.7563E05	is converted to	-675630.0

D CONVERSION

The *D* conversion is used for the input/output of double-precision numbers. It is used exactly as the *E* conversion except the letter E is replaced by D.

I CONVERSION

General form:

rIw

Only integer data may be processed by this form of conversion.

Output

Internal values are converted to integer constants. Negative values are preceded by a minus sign. Each field is right-justified and filled with leading blanks.

For a field specification of I6:

281	is converted to	281
-3567	is converted to	-3567

If the data require more character positions than allowed by the width, w, only the most significant w positions are output.

For a field specification of I3:

281	is converted to	3*
-6374	is converted to	-6*

Input

External input values are right-justified with the width, w. Blanks are counted as zeros. Input values must be integer values. A preceding minus sign may be placed on a value.

For a field specification of I4:

120	is converted to	120
-144	is converted to	-144
1 2	is converted to	102
-3	is converted to	-3

A CONVERSION

An *A* format conversion is used in conjunction with a *READ* or *WRITE* statement for the input/output of alphanumeric information to or from a **REAL**, **INTEGER**, or **LOGICAL** list element. The general form is rAw, where r and w are unsigned integer constants. If r is one, it can be omitted.

On input, rAw will be interpreted to mean that the next r successive fields of w characters are each to be stored in the asscoiated *REAL* list elements. If w is greater than 9, where 9 is the number of characters a single list element can contain, only the 9 right-most characters will be significant. If w is 9 or less, the characters will be left-justified, and the word(s) filled with blanks, if necessary.

On output, rAw will be interpreted to mean that the next r successive fields of w characters are each to be the result of alphanumeric transmission from the specified list elements. If w exceeds g, only g characters of output will be transmitted, preceded by w - g blanks. If w is g or less, the w left-most characters of the specified storage element will be transmitted.

H CONVERSION

In **FORTRAN**, Hollerith information consists of the legal **FORTRAN** character set plus the additional characters

" # : ; ! % & ' | [\ ] < > ?

Information input from the typewriter or paper tape is converted to an internal code used by **FORTRAN**. When this information is output, the internal codes are converted to the appropriate typewriter or paper tape codes.

General form:

nHs

Or:

's' (VORTEX only)

Output

The number of characters, n, in the string, s, should contain exactly the number of characters specified so that characters from other fields are not taken as part of the string.

Blanks are counted as characters in the string. The quote character (') can be output using a pair of quotes in the 's' format description.

Examples

Specification	**External Output**
1HR	R
8HbSTRINGb	bSTRINGb
11HX(1,3) = 12.0	X(1,3) = 12.0
'bA = '	bA =
'bs = " A"'	bs = 'A'

b indicates a blank space

Input

The w characters in the string, s, are replaced by the next w characters from the input record. The result is a new string in the field specification. Each quote in a pair is overlayed by an input character in the 's' format.

Example

Specification	**Input String**	**Resultant Specification**
5H12345	ABCDE	5HABCDE
7HbTRUEbb	FALSEbb	7HFALSEbb
8Hbbbbbbbb	MATRIXbb	8HMATRIXbb
'AB'	12	'12'
'X'	ABC	'ABC'

b indicates a blank space

This feature can be used to change titles, dates, headings, etc., that are output with the program data.

X SPECIFICATION

General form: nX.

This specification causes no conversion. On output, n blanks are inserted in the external record. On input, n spaces are skipped from the input record.

Output Example

Specification	Output
1HA, 4X, 2HBC	AbbbbBC
4X, 3HABC	bbbbABC
1X, 3HABC, 3X	bABCbbb

Input Example

Specification	Input String	Resultant Input
F4.1, 3X, F3.0	12.5RRR120	12.5,120.

The RRR characters are ignored by the 3X specification.

L FORMAT CODE

General form:

rLw

where:

r is optional and is an unsigned integer constant used to denote the number of times the same format code is repetitively referenced.

w is an unsigned integer constant that specifies the number of characters of data.

Logical variables may be read or written by means of the format code Lw.

On input, the first T or F encountered in the next w characters of the input record causes a value of .TRUE. or .FALSE., respectively, to be assigned to the corresponding logical variable. If field w consists entirely of blanks, a value of .FALSE. is assumed.

On output, a T or F is inserted in the output record as the value of the logical variable in the I/O list. T is a non-zero value and F is zero. The single character is preceded by w − 1 blanks.

G FORMAT CODE

General form:

rGw.d

where:

r is optional and is an unsigned integer constant used to denote the number of times the same format code is repetitively referenced.

w is an unsigned integer constant specifying the total field length.

d is an unsigned integer constant specifying the number of significant digits.

The G format code is a generalized code in that it automatically selects an output format appropriate to the magnitude of the real data.

Input processing is the same as for the F conversion.

The w portion of the G format code reserves the four right-most positions for a decimal exponent field.

If the real data, n, are in the range $0.1 \leq n < 10^{**}d$, where d is the d portion of the format code Gw.d, then this exponent field is blank. Otherwise, the real data are transferred with an E or D decimal exponent depending on the type of the real data.

For the purpose of simplification, the following examples deal with the printed line. However, the concepts apply to all input/output media.

Example 1

Assume that the variables A, B, C, and D are of type real whose values are 292.7041, 82.43441, 136.7632, 0.8081945, respectively.

```
1       FORMAT    (G12.4,G12.5,G12.4,G12.7)
2       FORMAT    (G13.4,G13.5,G13.4)
3       FORMAT    (G13.4)
          .
          .
          .
        WRITE     (0, n) A, B, C, D
          .
          .
          .
```

Explanation:

a. If n has been specified as 1, the printed output would be as follows (b represents a blank):

```
Print Position 1                                          Print Position 48
↑                                                         ↑
bbb292.7bbbbbb82.434bbbbbbb136.8bbbb.8081945bbbb
```

b. If n has been specified as 2, the printed output would be:

```
Print Position 1                                  Print Position 39
↑                                                 ↑
bbbb292.7bbbbbbb82.434bbbbbbbb136.8bbbb                    Line 1
bbbb.8082bbbb                                              Line 2
```

From the above example, it can be seen that by increasing the field width reserved (w), blanks are inserted.

c. If n has been specified as 3, the printed output would be:

```
Print Position 1
↑
bbbb292.7bbbb                                              Line 1
bbbb82.43bbbb                                              Line 2
bbbb136.8bbbb                                              Line 3
bbbb.8082bbbb                                              Line 4
```

From the above example, it can be seen that the same format code is used for each variable in the list. Each repetition of the same format code causes a new line to be printed.

T SPECIFICATION (VORTEX ONLY)

General form:

Ty

where:

T is a specification that relocates the current position in the external record.

y is a non-zero integer constant that specifies the character position in the external record.

On output, a T specification can be used to position column headers as follows (b indicates a blank space):

```
1     FORMAT(T10,5HCOLb1 , T22 , 5HCOLb2)
```

This example causes

```
COLb1
```

to be printed starting in column 10, and causes

```
COLb2
```

to be printed starting in column 22.

On input, a T specification can be used to skip or re-read fields.

SCALE FACTOR P

The representation of the data, internally or externally, can be modified by the use of a scale factor followed by the letter P preceding the F, E, G, and D format codes.

The scale factor affects the appropriate conversions in the following manner:

a. For F, E, G, and D input conversions (provided no exponent exists in the external field) and F output conversions, the scale factor effect is as follows:

   externally represented number equals internally represented number times the quantity ten raised to the nth power.

b. For F, E, G, and D input, the scale factor has no effect if there is an exponent in the external field.

c. For E and D output, the basic real constant part of the quantity is multiplied by ten to the nth power and the exponent is reduced by the scale factor.

d. For G output, the effect of the scale factor is suspended unless the magnitude of the datum to be converted is outside the range that permits the effective use of F conversion. If the effective use of E conversion is required, the scale factor has the same effect as with E output.

For example, if input data are in the form xx.xxxx and it is desired to use this internally in the form .xxxxxx, the format code used to effect this change is 2PF7.4.

Input

As another example, consider the following input data:

27bbb-93.2094bb-175.8041bbbb55.3647

where b represents a blank.

The following statements:

```
5       FORMAT      (I2,3F11.4)
          .
          .
          .
        READ        (0,5) K,A,B,C
```

cause the variables in the list to assume the following values:

```
K : 27              B : -175.8041
A : -93.2094        C : 55.3647
```

The following statements:

```
5       FORMAT      (I2,1P3F11.4)
          .
          .
          .
        READ        (0,5) K,A,B,C
```

cause the variables in the list to assume the following values:

```
K : 27              B : -17.58041
A : -9.32094        C : 5.53647
```

The following statements:

```
5       FORMAT      (I2,-1P3F11.4)
          .
          .
          .
        READ        (0,5) K,A,B,C
```

causes the variables in the list to assume the following values:

```
K : 27              B : -1758.041
A : -932.094        C : 553.647
```

Output

Assume the variables K,A,B, and C have the following values:

```
K : 27          B : -175.8041
A : -93.2094    C : 55.3647
```

then the following statements:

```
5       FORMAT      (I2,1P3F11.4)
          .
          .
          .
        WRITE       (0,5) K,A,B,C
```

cause the variables in the list to output the following values:

```
K : 27          B : -1758.041
A : -932.094    C : 553.647
```

The following statements:

```
5       FORMAT      (I2,-1P3F11.4)
          .
          .
          .
        WRITE       (0,5) K,A,B,C
```

cause the variables in the list to output the following values:

```
K : 27          B : -17.5804
A : -9.3209     C : 5.5365
```

For output, when scale factors are used, they have effect only on real data. However, this real data may contain an E or D decimal exponent. A positive scale factor used with real data that contains an E or D decimal exponent increases the number and decreases the exponent. Thus, if the real data were in a form using an E decimal exponent and the statement FORMAT (1X,I2,3E13.3) used with an appropriate *WRITE* statement resulted in the following printed line:

b27bbbb-.932Eb02bbbb-.175Eb03bbbbb.553Eb02

the statement FORMAT (1X,I2,1P3E13.3) used with the same *WRITE* statement results in the following printed output:

b27bbb-9.321Eb01bbb-1.758Eb02bbbb5.536Eb01

The statement **FORMAT** (1X,I2,-1P3E13.3) used with the same **WRITE** statement results in the following printed output:

27bbbb-.093Eb03bbbb-.018Eb04bbbbb.055EB05

The scale factor is assumed to be zero if no other value has been given. However, once a value has been given, it will hold for all format codes following the scale factor within the same *FORMAT* statement. This also applies to format codes enclosed within an additional pair of parentheses.

/ SPECIFICATION

Form

Each slash (/) specified in the format causes the termination of a record and processing of the next record. Successive slashes (///...//) cause subsequent records to be ignored on input, and successive blank records to be written on output. A slash separating two field specifications removes the need for a comma separator. For example,

F5.4/4F10.3.

Output Example

For a specification (1HA/1HB/1HC/1HD) the resultant output records are:

A
B
C
D

Input Example

Using the four records output from the previous example, an input specification (1H1/1H2//1H3) produces the resultant specification (1HA/1HB//1HD).

REPEAT SPECIFICATIONS

The A, D, F, E, I, L, and G field specifications can be repeated by using the repeat count r in the forms rAw, rDw, rFw.d, rEw.d, rIw, rLw, and rGw.d.

Examples

4F10.5,F3.6 is equivalent to F10.5,F10.5,F10.5,F10.5,F3.6

2F4.1,2E7.1 is equivalent to F4.1,F4.1,E7.1,E7.1

2F5.2,3I6,2E8.2 is equivalent to F5.2,F5.2,I6,I6,I6,E8.2,E8.2

Repetition of a group of field specifications is accomplished by enclosing the group in parentheses preceded by an integer repeat count. If no repeat count is specified, the count is taken as one.

Examples

2(F10.5,I6) is equivalent to F10.5,I6,F10.5,I6

2(E9.3,F7.1/I4) is equivalent to E9.3,F7.1/I4,E9.3,F7.1/I4

3(4F5.0,2E8.2) is equivalent to 4F5.0,2E8.2,4F5.0,2E8.2,
4F5.0,2E8.2

Example

```
50 FORMAT (4X,2(I5,6F8.2)//)
```

The use of additional parentheses (up to two levels) within a *FORMAT* statement is permitted to enable the user to repeat the same format code when transmitting data. For example, the statement:

```
10 FORMAT (2(G10.6,G7.1),G4)
```

is equivalent to

```
10 FORMAT (G10.6,G7.1,G10.6,G7.1,G4)
```

If the data exists with a D decimal exponent, it is transferred with this D decimal exponent.

If a multiline listing is desired such that the first two lines are to be printed according to a special format and all remaining lines according to another format, the last format code in the statement should be enclosed in a second pair of parentheses. For example, in the statement:

```
FORMAT(G2,2G3.1/G10.8/(3G5.1))
```

If more data items are to be transmitted after the format codes have been completely used, the format repeats from the last left parenthesis. Thus, the printed output would take the following form:

```
G2,G3.1,G3.1
G10.8
G5.1,G5.1,G5.1
G5.1,G5.1,G5.1
.
.
.
```

As another example, consider the following statement:

```
FORMAT(G2/2(G3,G6.1),G9.7)
```

If 13 data items are to be transmitted, the printed output on a *WRITE* statement takes the following form:

```
G2
G3,G6.1,G3,G6.1,G9.7
G3,G6.1,G3,G6.1,G9.7
G3,G6.1
```

FORMAT CONTROL AND LINE INTERACTION

Execution of a formatted *READ* or *WRITE* statement initiates format control. The conversion performed on data depends on information jointly provided by the next element of the input-output list and the next field specification of the *FORMAT* statement. If there is a list, at least one field specification of type D, E, F, G, L, A, or I should be present in the *FORMAT* statement.

Execution of a formatted *READ* statement causes one record to be input. Each D, E, F, G, L, A, or I specification has a corresponding element in the list. Each H or X specification has no corresponding element in the list and the format control communicates information directly to the record. When a slash is encountered or the entire input record is processed, the record is terminated. If more input is necessary, the next record is input. Any unprocessed characters of a record are skipped when a slash is encountered.

A *READ* statement is terminated upon ending the list if:

a. The next specification is A, D, E, F, G, I, or L.

b. The format control has reached the last outer right parenthesis of the FORMAT statement.

If the list ends and the next specification is an H or X, data are processed (with the possibility of additional records being input) until one of the two above conditions is met.

If the format control reaches the right-most parenthesis of the *FORMAT* statement and more list remains to be processed, the following steps are taken:

a. A new record is input and remaining data in the previous record ignored.

b. Format control reverts to the point immediately following the last left parenthesis.

If group repeat specifications exist in the format, this point is at the right-most group of the format. The repeat count is not taken into consideration. If no groups are present, the format is started from the beginning.

When a formatted *WRITE* statement is executed, records are written each time 120 characters have been processed, a slash is encountered, or the format control terminates. The format control terminates by one of the two methods described for *READ* termination. Incomplete records are filled with blanks to maintain standard record lengths.

COMMA AS DELIMITER ON INPUT

Varian 73/620 **FORTRAN** allows a comma to be used as a delimiter between inputs. As an example with the following format statement:

```
FORMAT (I3, I4, F6.2)
```

Using the above format, the following values can be input with a READ statement: 13, 2, 12.60

The run-time I/O accepts this input and handles these values correctly, however you cannot input more than the number of characters specified in the FORMAT statement.

SECTION 7— PROGRAMS AND SUBPROGRAMS

An executable **FORTRAN** program consists of a main program and any required subprograms. Subprograms may be defined by the programmer or contained in the system library. Each program or procedure subprogram must contain at least one executable statement.

Each VORTEX program or subprogram can contain as its first statement (except for comment lines) a TITLE statement with the following format:

TITLE n

where:

n is the program module name that is included in the heading of the source listing, as well as in the object program used by system maintenance and generation programs in VORTEX.

MAIN PROGRAMS

A main program is a program unit consisting of a set of **FORTRAN** statements, comment lines, and an *END* line. The program may be preceded by specification statements.

A main program cannot contain a subprogram definition statement, namely:

a FUNCTION statement
a SUBROUTINE statement
a BLOCK DATA statement

A main program may contain calls to other subprograms or may contain statement function subprograms.

A main program can accept a main program entry name definition of the following format:

NAME N1, N2, ..., Nn

where:

N1, N2, ..., Nn are entry names by which the main program can be referenced

A VORTEX main program must include specifications for all common blocks that are referenced by the subprograms.

SUBPROGRAMS

Subprograms are program units which may be called by other programs or subprograms. Subprograms are categorized as one of the following:

PROCEDURE SUBPROGRAMS
- FUNCTION subprogram
- SUBROUTINE subprogram

SPECIFICATION subprogram
- BLOCK DATA subprogram

Functions are programmed procedures that are often used to provide solutions to mathematical functions. Function references may be used in the same manner as references to variables in an expression. For example: X = AB*SIN (Y) - C*COS (Y*Z), where SIN is the name of the sine function, COS is the name of the cosine function, and (Y) and (Y*Z) are their respective argument lists. The value returned for a function reference is of the same mode as the function name, corresponding to the rules for real and integer symbolic names.

Function Subprograms

A function subprogram is defined external to the program unit by which it is referenced. A function subprogram is defined by having as its first statement, other than comment lines, a statement of the form:

FUNCTION f(a1, a2, a3, ..., an)

where

f is the symbolic name of the function and

ai represent dummy arguments.

Each ai is either a variable name, array name, or an external procedure name. The ai defines the type, number, and order of the *FUNCTION* arguments. A function subprogram must have at least one argument.

A function subprogram is executed at the first executable statement following the *FUNCTION* statement. Specification statements (*DIMENSION*, *COMMON*, and *EQUIVALENCE*) may immediately follow the *FUNCTION* statement. If present, these must precede any other statement, excluding comments. The symbolic names of the dummy arguments, ai, may not appear in an *EQUIVALENCE* or *COMMON* statement.

A function subprogram must contain at least one *RETURN* statement, and the last statement executed in a *FUNCTION* must be a *RETURN* statement. The function subprogram is ended by an *END* line.

The symbolic name, f, of the *FUNCTION* must appear as a variable name within the subprogram. The value returned for a *FUNCTION* is the last value assigned to this name prior to execution of a *RETURN* statement. The type of the *FUNCTION* value is as for a variable (section 2.3.1).

The symbolic name of the function must not appear in any nonexecutable statement within the subprogram.

Example

```
      FUNCTION XP(A,B,I)
      DIMENSION B(10)
      XP = 0.
      DO 1 J = 1,10
1     XP = (A*B(J))**I + XP
      RETURN
      END
```

A *FUNCTION* is executed with a function reference by a main program or another subprogram. The actual arguments in the call must correspond in type, number, and order with the *FUNCTION* dummy arguments. If a dummy argument of a *FUNCTION* is an array name, the corresponding actual argument must be an array name.

Example:

A call for the example *FUNCTION* shown above would be: W = XP(R,S,K), where S is an array.

Type Specification of FUNCTION Subprogram

In addition to declaring the type of a *FUNCTION* name by the predefined convention, there exists the option of explicitly specifying the type of a *FUNCTION* name within the function statement.

General Form

Type *FUNCTION* name (a1, a2, a3, ..., an)

where:

Type is integer, real, double precision, complex, or logical.

name is the name of the *FUNCTION* subprogram.

a1, a2, a3,...,an are nonsubscripted variables, or dummy names of subroutine or other FUNCTION subprograms. (There must be at least one argument in the argument list.)

Example 1

```
REAL FUNCTION SOMEF (A,B)
  .
  .
  .
SOMEF = A**2 + B**2
  .
  .
  .
RETURN
END
```

Example 2

```
INTEGER FUNCTION CALC (X,Y,Z)
  .
  .
  .
CALC = X + Y + Z**2
  .
  .
  .
RETURN
END
```

Explanation:

The *FUNCTION* subprograms SOMEF and CALC in Examples 1 and 2 are declared as type *REAL* and *INTEGER*, respectively.

Subroutine Subprograms

A subroutine subprogram is defined external to the program unit that references it. Subroutines, unlike functions, do not have values associated with them and cannot be referenced in an expression. Subroutines are accessed by *CALL* statements.

A subroutine subprogram is defined by having as its first statement, other than comment lines, a statement of the form: SUBROUTINE S (a1, a2, a3, ..., an) or SUBROUTINE S, where S is the symbolic name of the subroutine and ai represents dummy arguments of the subroutine. Each ai is either a variable or an array name, or the name of an external procedure. If no arguments are passed to the subroutine, the second form is used.

The symbolic name of the subroutine must not appear in any statement in the subprogram. The symbolic names of the dummy arguments may not appear in *COMMON* or *EQUIVALENCE* statements.

A subroutine is executed at the first executable statement. Specification statements must immediately follow the *SUBROUTINE* statement and precede any executable statement. A

subroutine must have at least one *RETURN* statement. The last statement executed by a subroutine must be a *RETURN* statement.

Example

```
      SUBROUTINE R(A,I,Z)
      DIMENSION A(10)
      Z=0
      DO 1 J=1,10
1     Z=Z+A(J)**I
      RETURN
      END
```

A subroutine is referenced with a *CALL* statement. The argument list in the reference must agree in type, number, and order with the dummy arguments of the subroutine. If a dummy argument is an array name, the corresponding actual argument must be an array name.

Example:

A call for the example SUBROUTINE above would be: CALL R (T,K,D) where T is an array.

Block Data Subprogram

To intialize variables in a *COMMON* block, a separate subprogram must be written. This separate subprogram contains only the *DATA*, *COMMON*, *DIMENSION*, *EQUIVALENCE*, and *TYPE* statements associated with the data being defined. In the MOS and stand-alone systems, *COMMON* blocks are assigned downward from the top of available memory, with the blank block first and the others in the order they appear in the source text. The loader is overlaid by these block; therefore, when using the *BLOCK DATA* statement, the programmer must be aware of the block location to prevent data from being stored in loader tables.

General Form

```
BLOCK DATA
  .
  .
  .
END
```

a. The *BLOCK DATA* subprogram may not contain any executable statements.

b. The *BLOCK DATA* statement must be the first statement in the subprogram.

c. All elements of a COMMON block must be listed in the *COMMON* statement, even though they are not all initialized; for example, the variable A in the *COMMON*

statement in the following example does not appear in the data initialization statement.

```
BLOCK DATA
COMPLEX C
COMMON/ELN/C,A,B/RMG/Z,Y
DATA C/(2.4.3.769)/
```

d. Data may be entered into more than one COMMON block in a single *BLOCK DATA* subprogram.

e. An optional entry name n can follow the *BLOCK DATA* statement:

```
BLOCK DATA n
```

This causes output of n as an entry name so that the subprogram can be stored in a library enabling it to be loaded with any module containing an EXTERNAL n statement.

Data Initialization Statement

General Form

DATA k1,...,kn/j1*d1,...,jn*dn/,kn + 1,...,k/jn + 1*dn + 1,...,j*d/,...

where:

k1,...,k are variables and/or subscripted variables (in which case the subscripts must be integer constants), or array names, or implied DO lists

d1,...,d are values representing integer, real, double-precision, complex, logical or Hollerith data constants.

j1*,...,j* represent unsigned integer constants indicating the number of consecutive variables that are to be assigned the value of d1,...,dn.

A data initialization statement defines initial values of variables and array elements. There must be a one-to-one correspondence between these variables (i.e., k1,..., k) and the data constants (i.e., d1,..., d).

Example 1

```
DIMENSION D(10)

DATA A,B,C/5.0,6.1,7.3/,D(1),D(2),D(3),D(4),D(5)/5*1.0/
```

Explanation:

The *DATA* statement indicates that the variables A, B, and C are to be initialized to the values 5.0, 6.1, and 7.3, respectively. In addition, the statement specifies that the first five variables in array D are to be initialized to 1.0.

Example 2

```
DIMENSION A(5),B(3),L(2)

DATA A(1),A(2),A(3),A(4),A(5)/5*1.0/,B(1),B(2)/2*5.0/,L(1),L(2)/.TRUE.,.FALSE./
```

Explanation:

The *DATA* statement specifies that all the variables in array A are to be initialized to 1.0 and the first two elements of array B are to be initialized to 5.0. The logical variables, (L(1) and L(2)), in array L are initialized to .TRUE. and .FALSE., respectively.

An initially defined variable, or any element, may not be in blank common. However, in a labeled *COMMON* block, they may be initially defined only in a block data subprogram. (See the Subprograms section.)

Example 3

```
DIMENSION A(3), B(3,2)
DATA A/1.0,2.0,3.0/,((B(I,J),J = 1,2),I = 1,3)/6*5./
```

Explanation:

The *DATA* statement loads real numbers 1.0, 2.0, and 3.0 into array A. It also loads real number 5. into every element of array B. DATA statements must precede the first executable statement or statement function, and must follow any specification statements.

STATEMENT FUNCTIONS

A statement function is defined internal to the program unit in which it is referenced. All statement functions must precede the first executable statement and must follow any specification statements of the program unit.

A statement function is defined in a single expression of the form: f(a1, a2, a3, ..., an) = e, where f is the function name, ai represents arguments, and e is an expression. The resultant value of the function is either a real or integer value corresponding to the function name. The ai are distinct variable names and are called dummy arguments. They serve to indicate the type, number, and order of the function arguments. The expression e is an arithmetic expression and may contain references to previously defined statement functions.

A statement function is referenced by a function call, f(a1, a2, a3, ..., an), appearing in an arithmetic expression. A statement function may be referenced only within the program unit in which it is defined. The arguments used in the reference must agree in type, number, and order with the corresponding dummy arguments.

Example

The statement function:

```
SF(X) = A*X**2+B*X+C
```

can be referenced in the program by:

```
W = SF(Y)
```

INTRINSIC FUNCTIONS

Intrinsic functions are commonly used subprograms contained in the **FORTRAN** library. The symbolic names and meanings of the intrinsic functions are shown in table 7-1.

An intrinsic function is referenced by a function call in an arithmetic expression. The arguments in the argument list must agree in type, number, and order with those shown in table 7-1.

Example

```
      IF (SIGN(W,X)) 1,2,2
1     W = ABS(X)-ABS(Y)
2     S = W*FLOAT(I*J)
      K = IFIX(X)+J
```

BASIC EXTERNAL FUNCTIONS

Basic external *FUNCTIONS* are standard subprograms contained in the **FORTRAN** library. These are referenced in the same manner as normal *FUNCTIONS*. The symbolic names and meanings of the basic external *FUNCTIONS* are shown in table 7-2.

DUMMY ARGUMENTS

Dummy arguments provide a means of passing information between a subprogram and the program or subprogram that called it. Both function and subroutine subprograms may have dummy arguments. A subroutine need not have any, while a function must have at least one. Dummies provide definitions of the data type, number, and sequence of subprogram parameters.

A dummy can be classified within a subprogram as a variable, an array, or an external procedure name. The actual arguments defined by a calling program or subprogram to which a dummy can correspond are: Hollerith constants, variables, array elements, arrays, expressions, and external procedure names.

Within a subprogram, a dummy can be used in much the same way as any other variable or array. A dummy can not appear in a *COMMON* or *EQUIVALENCE* statement.

The actual arguments (except for Hollerith constants) used in a calling statement agree in data type with the corresponding dummy arguments, that is, real to real, integer to integer, and array to array. If an actual argument is an expression, the result of the expression should correspond in data type to the dummy.

A dummy array is defined as an argument which appears in a *DIMENSION* statement in the subprogram. A dummy array does not occupy any storage but tells the subprogram that the argument supplied in the calling statement defines the first element of an actual array. The calling argument need not have the same dimensions as the dummy array. Useful operations can sometimes be performed by defining different dimensions for the dummy and calling arguments.

Example

```
DIMENSION       A(10,10)
CALL            FM(A(6,1))
   .
   .
   .
SUBROUTINE      FM(B)
DIMENSION       B(50)
```

For this case, one-dimensional dummy array B corresponds to the last half of two-dimensional array A. If the calling statement were CALL FM (A), dummy array B would correspond to the first half of array A.

ADJUSTABLE DIMENSIONS

As shown in the previous examples, the maximum value of each subscript in an array is specified by a numeric value. These numeric values (maximum value of each subscript) are known as the absolute dimensions of an array and may never be changed. However, if an array is used in a subprogram (section 7.3) and is not in *COMMON*, the size of this array does not have to be explicitly declared in the subprogram by a numeric value. That is, the specification statement, appearing in a subprogram, may contain integer variables that specify the size of the array. These integer variables must be either actual or implicit subprogram arguments. When the subprogram is called, these integer variables receive their values from the calling program. Thus, the dimensions (size) of a dummy array appearing in a subprogram are adjustable and may change each time the subprogram is called. Integer variables that provide dimension information may not be redefined within the subprogram.

The absolute dimensions of an array must be declared in a calling program. The adjustable dimensions of an array, appearing in a subprogram, should be less than or equal to the absolute dimensions of that array as declared in the calling program.

The following example illustrates the use of adjustable dimensions.

Example

CALLING PROGRAM

```
DIMENSION A(5,5)
      .
      .
      .
CALL MAPMY(...,A,2,3,...)
      .
      .
      .
```

SUBPROGRAM

```
SUBROUTINE MAPMY
(...,R,L,M,...)
      .
      .
      .
DIMENSION...,R(L,M),...
      .
      .
      .
DO 100 I = 1,L
      .
      .
      .
```

Explanation:

The statement DIMENSION A(5,5) appearing in the calling program declares the absolute dimensions of array A. When subroutine MAPMY is called, dummy argument R assumes array name A and dummy arguments L and M assume the values 2 and 3, respectively. The correspondence between the subscripted variables of arrays A and R is shown in the following example.

R(1,1)R(2,1)R(1,2)R(2,2)R(1,3)R(2,3)

A(1,1)A(2,1)A(3,1)A(4,1)A(5,1)A(1,2)A(2,2)...

Thus, in the calling program the subscripted variable A(1,2) refers to the sixth subscripted variable in array A. However, in subprogram MAPMY, the subscripted variable R(1,2) refers to the third subscripted variable in array A, namely, A(3,1). This is so because the dimensions of array R as declared in the subprogram are not the same as those in the calling program.

If the absolute dimensions in the calling program were the same as the adjusted dimensions in the subprogram, the subscripted variables R(1,1) through R(5,5) in the subprogram would always refer to the same storage locations as specified by the subscripted variables A(1,1) through A(5,5) in the calling program, respectively.

The numbers 2 and 3, which became the adjusted dimension of dummy array R, could also have been variables in the argument list or implicit arguments in a *COMMON* block. For example, assume that the following statement appeared in the calling program.

```
CALL MAPMY (...,A,I,J,...)
```

Then as long as the values of I and J are previously defined, the arguments may be variables. In addition, the variable dimension size may be passed through more than one subprogram level. For example, the subprogram MAPMY could have contained a call statement to another subprogram in which dimension information about A could have been passed.

Dummy variables (e.g., L and M) may be used as dimensions of an array only in a *FUNCTION* or *SUBROUTINE* subprogram.

EXTERNAL STATEMENT

When an actual parameter list of a function reference or a subroutine call contains a function or subroutine name, that name must appear in an EXTERNAL statement in the program in which the reference or call appears.

The form of the EXTERNAL statement is

```
EXTERNAL  s, s, s,... s
```

where s is a function or subroutine name

The EXTERNAL statement must appear before the function or subroutine reference. A statement function may not appear in an EXTERNAL statement.

The following are examples of valid EXTERNAL statements

```
EXTERNAL  SUB1, SINF
EXTERNAL  FRAIL
```

Table 7-1. Intrinsic Functions

Intrinsic Function	Definition	Arguments	Name	Type of Argument	Type of Function
Absolute Value	$\|a\|$	1	ABS	Real	Real
			IABS	Integer	Integer
			DABS	Double	Double
Truncation	Sign of a times largest integer $\leq \|a\|$	1	AINT	Real	Real
			INT	Real	Integer
			IDINT	Double	Integer
Remaindering*	$a_1 \pmod{a_2}$	2	AMOD	Real	Real
			MOD	Integer	Integer
Choosing Largest Value	Max $(a_1, a_2, \ldots)$	≥ 2	AMAX0	Integer	Real
			AMAX1	Real	Real
			MAX0	Integer	Integer
			MAX1	Real	Integer
			DMAX1	Double	Double
Choosing Smallest Value	Min $(a_1, a_2, \ldots)$	≥ 2	AMIN0	Integer	Real
			AMIN1	Real	Real
			MIN0	Integer	Integer
			MIN1	Real	Integer
			DMIN1	Double	Double
Float	Conversion from integer to real	1	FLOAT	Integer	Real
Fix	Conversion from real to integer	1	IFIX	Real	Integer

Transfer of Sign	Sign of a_2 times $\|a_1\|$	2	SIGN ISIGN DSIGN	Real Integer Double	Real Integer Double
Positive Difference	a_1 - min (a_1 , a_2)	2	DIM IDIM	Real Integer	Real Integer
Obtain Most Significant Part of Double-Precision Argument		1	SNGL	Double	Real
Obtain Real Part of Complex Argument		1	REAL	Complex	Real
Obtain Imaginary Part of Complex Argument		1	AIMAG	Complex	Real
Express Single-Precision Argument in Double-Precision Form		1	DBLE	Real	Double
Express Two Real Arguments in Complex Form	$a_1 + a_2\sqrt{-1}$	2	CMPLX	Real	Complex
Obtain Conjugate of a Complex Argument		1	Conjg	Complex	Complex

*The function MOD or AMOD (a_1 , a_2) is defined as a_1 - [a_1 /a_2] a_2 , where [x] is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as x.

Table 7-2. Basic External Functions

External Functions	Definition	Arguments	Name	Type of Argument	Type of Function
Exponential	e^a	1	EXP	Real	Real
			DEXP	Double	Double
			CEXP	Complex	Complex
Natural Logarithm	$\log_e (a)$	1	ALOG	Real	Real
			DLOG	Double	Double
			CLOG	Complex	Complex
Common Logarithm	$\log_{10} (a)$	1	ALOG10	Real	Real
			DLOG10	Double	Double
Trigonometric Sine	sin(a)	1	SIN	Real	Real
			DSIN	Double	Double
			CSIN	Complex	Complex
Trigonometric Cosine	cos(a)	1	COS	Real	Real
			DCOS	Double	Double
			CCOS	Complex	Complex
Hyperbolic Tangent	tanh(a)	1	TANH	Real	Real
Square Root	$(a)^{1/2}$	1	SQRT	Real	Real
			DSQRT	Double	Double
			CSQRT	Complex	Complex
Arctangent	arctan(a)	1	ATAN	Real	Real
			DATAN	Double	Double
	$\arctan(a_1 / a_2)$	2	ATAN2	Real	Real
			DATAN2	Double	Double
Remaindering*	$a_1 \pmod{a_2}$	2	DMOD	Double	Double
Modulus		1	CABS	Complex	Real

*The function DMOD (a_1, a_2) is defined as $a_1 - [a_1 / a_2] a_2$, where [x] is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as the sign of x.

COMBINING FORTRAN AND DAS MR

FORTRAN generates the following calling sequence for all implicit and explicit calls to subprograms:

```
JUMP    s
DATA    P1
DATA    P2
 .      .
 .      .
 .      .
DATA    Pn
```

where

s is the subprogram name

n is the number of arguments

P1, P2, and Pn are the addresses (not the value) of the arguments; these addresses can be direct or indirect.

If the above calling sequence is used, DAS MR programs can reference any program in the system library or any **FORTRAN** coded subprogram.

DAS MR subprograms to be used with **FORTRAN** must process the above calling sequence. The library program $SE can be used to transfer parameters by coding the DAS MR subprogram entry as follows:

```
s       ENTER
        CALL $SE
        DATA n
        BSS n
```

where

s is the subprogram name

n is the parameter count

$SE transfers the n parameter addresses, resolving indirect addresses sequentially into the block defined by BSS n. In addition, $SE increments the address in s so that the program returns to the address following the calling sequence.

The above calling sequence does not define a parameter count so it is difficult to use with subprograms that process a variable-length parameter list. The only library programs of this type are the intrinsic functions that list maximum and minimum values. The **FORTRAN** compiler detects calls to these values and outputs an absolute zero to mark the end of the parameter list. DAS MR programs can reference these functions by terminating the calling sequence with an absolute zero (not a pointer to zero).

SECTION 8 – STAND-ALONE OPERATING PROCEDURES

The **Stand-Alone FORTRAN/DAS MR** system is a FORTRAN IV compiler and macro assembler (DAS MR) with loader and math support routines. Using a minimum system configuration containing 8K of memory and a 33/35 ASR Teletype,* the stand-alone system can generate relocatable binary output, and can load, link, and execute this output. In addition, the generalized I/O structure defines peripherals at loading time.

CONFIGURATION

The stand-alone system is contained on 12 separate paper tapes. Table 1 lists the tapes in numerical order along with their format and function. The various subroutines contained in each of the support libraries on tape 5 through 12 are listed below in order as they appear on each tape.

Tape 5. Run-time I/O

FORTIO	CRIE	MT$3	TCK$
$00	$0Q ($OR)	MTAE	$TC01
$04	$0Q	KNT$	$HC37
$08	$0P	RDC$	HCK$
$0C	$0S	WRT$	DIM$
$0G	CPAE	STR$	LAS$
$0H/$01	MT$0	SWR$	IOA$
$00	MT$1	BL$P	IOOK
$0M	MT$2	FCH$	$BICD

Tape 6. Run-time utility

$DO	$SE	RSCB3
$CG	FORTUTIL	RSCBIMTB
$3S	$EE	$BUF

Tape 7. Run-time math, software M/D, single precision

$HE	SINCOS	XDADD	ISIGN
$PE	FMULDIV	XDSUB	SIGN
$QE	FADDSUB	XDCOMP	$HN-S
ALOG	SEPMANTI	$FLOAT	$HM-S
EXP	FNORMAL	$IFIX	XMUL
ATAN	XDDIV-S	IABS	XDIV
SQRT-S	XDMULT-S	ABS	I$FA

* A preferable basic system is 12K of memory and a high-speed paper tape reader in addition to the Teletype.

Tape 8. Run-time math, hardware M/D, single precision

$HE	SINCOS	XDADD	ISIGN
$PE	FMULDIV	XDSUB	SIGN
$QE	FADDSUB	XECOMP	$HN-H
ALOG	SEPMANTI	$FLOAT	$HM-H
EXP	FNORMAL	$IFIX	XMUL
ATAN	XDDIV-H	IABS	XDIV
SQRT-H	XDMULT-H	ABS	I$FA

Tape 9. Run-time math, double precision

DSINCOS	CHEB	DADDSUB	DOUBLE
DATAN	DSQRT	DNORMAL	DEBLECOMP
DEXP	$DFR	DLOADAC	AC
DLOG	IDINT	DSTOREAC	
IF	DMULT	RLOADAC	
POLY	DDIVIDE	SINGLE	

Tape 10. Complex math functions

$9E	CABS	$AC	$ZD
CCOS	CONJG	CMPLX	AIMAG
CSIN	$AK	$8K	$OC
CLOG	$AL	$8L	REAL
CEXP	$AM	$8M	$8F
CSQRT	$AN	$8N	$8S

Tape 11. Math functions, single precision

TANH	AMAX1	MAX0	IDIM
ATAN2	AMIN0	MAX1	IFIX
ALOG10	AMIN1	MIN0	$JC
AMOD	DIM	MIN1	
AINT	FLOAT	MOD	
AMAX0	SNGL	INT	

Tape 12. Math functions, double precision

$XE	DMOD	DSIGN	DBLE
$YE	DINT	$YK	$XC
$ZE	DABS	$YL	
DATAN2	DMAXI	$YM	
DLOGIO	DMINI	$YN	

Table 1. Stand-Alone Tapes

Tape Number	Format	Function
1	BLD object	FORTRAN IV compiler, is listed on Teletype
2	BLD object	FORTRAN IV compiler, is listed on 620-77 line printer
3	MOS object (see MOS section)	DAS MR assembler
4	BLD object	Relocatable loader, loads tapes in MOS object format
5	MOS object	Run-time I/O
6	MOS object	Run-time utility
7	MOS object	Run-time math, software multiply/divide, single precision
8	MOS object	Run-time math, hardware multiply/divide, single precision
9	MOS object	Run-time math, double precision
10	MOS object	Complex math functions
11	MOS object	Math functions, single precision
12	MOS object	Math functions, double precision

MOS FUNCTIONS

Since the stand-alone system is a subset of MOS, various MOS functions also apply to this system. The following MOS items are applicable for the stand-alone system:

- I/O Calls
- Support Library
- User-Coded I/O Drivers
- Object Module Format
- Data Format

COMPILING A PROGRAM

The stand-alone FORTRAN IV compiler is supplied as an BLD object tape (tape 1 or 2). The compiler is loaded into memory by the BLD II and occupies approximately 017400 words of memory. Before loading the compiler, the SENSE switches on the computer control panel should be set according to the following options:

a. With no SENSE switches set, both BLD II and AID II programs are preserved.

b. With SENSE switch 1 set, only the BLD II program is preserved.

c. With SENSE switch 2 set, the last 17 words of memory are preserved (the last 17 memory locations normally contains the bootstrap loader).

The compiler is entered at location 0. It initializes itself, outputs the string 'DATE =' and inputs an 8-character date string in the form DD/MM/YY, which will appear on the source listing. It then inputs a 2-character compiler I/O specificaton (defined in a later paragraph) and halts with P = 3 and B = largest address used by the compiler. B can be manually modified at this time. The A register contents will be stored in the Processor Control Word ($PCW), so bits may be manually set in it according to the format listed in table 2.

Upon pressing RUN, the compiler will input source records. If character 1 of the first source record is a '/', this will be processed as a compiler control record: characters 7-14 will be stored as an 8-character program name, which will be output on the source listing; characters from 16 to the first blank will be decoded as Processor Control Word flags (table 2).

EXAMPLE:

The statement

```
/JOB   PROGRAM1 B9
```

causes the name PROGRAM1 to be output to the source listing device, and causes the compiler to suppress binary output and read the input as 029 (EBCDIC code).

Table 2. Processor Control Word Bits

Character	$PCW Bit	Function
B	1	Suppress binary output.
D	8	Allocate 2 words for integer and logical items.
E	14	Suppress End-of-File on binary output device after compilation.
L	0	Suppress list output.
S	2	Suppress post-compilation map.
X	6	Compile statements with first character 'X'.
9	15	Flag card input as in 029 (EBCDIC) code.

I/O Device Specifications

For each program to be compiled an = will be typed on the Teletype printer requesting input/output selection. The operator must respond by typing one of the following characters to indicate the input device:

C	Card reader
K	Teletype keyboard
P	High-speed paper-tape reader
T	Teletype paper-tape read
0-3	Magnetic tape unit # (device addresses 010 through 013)

followed by one of the following characters to indicate the output device:

P-T	Paper tape
0-3	Magnetic tape unit # (device addresses 010 through 013)

Following initialization, source statements are read and object records are output through the selected devices. Error diagnostics and selected list options are printed on the Teletype or printer (if available). Upon detection of an END statement, the compiler produces a program map listing all variables, constants (in octal), and required subprograms.

Compiler Input Records

Input to the compiler is a series of FORTRAN statements, each of which appears in one or more input records. Records can be fixed or variable length depending on the device; however, only the first 72 characters of each record are used by the compiler.

Keyboard and paper tape records are variable length and are terminated by a carriage return and line feed in that order. The character > can be used to TAB to column 7, and the character ← can be used to clear the input buffer and reset to column 1. For keyboard input the Teletype bell is used to notify the operator that source input is required. Paper tape leader must be less than 72 characters and terminate in ←. Source records input on the Teletype paper tape must be separated by ←←←

Card records are a fixed length of 80 characters. Magnetic tape records must be card images.

Compiler Output Records

Object records are output as they are created. Paper-tape object programs are punched with leader and trailer records.

Notification Errors

Errors are logged during both compilation and execution.

All compilation error diagnostics are of the form

ERR xx a...a

where

xx is a number from 0 to 18 (notification error), or T0 to T9 (termination error)

a...a represents the last (up to 12) characters encountered in the statement being processed.

The rightmost character indicates the point where the error was discovered (the character ← indicates end of the statement). If a termination error is discovered, object output is terminated, but source code is continued to detect any further errors.

The compilation error messages are:

1 Construction
2 Usage
3 Mode
4 Illegal DO termination
5 Improper statement number
6 COMMON base lowered
7 Illegal equivalence group
8 Reference to nonexecutable statement
9 No path to this statement
10 Multiply defined statement number
11 Invalid format construction
12 Spelling error
13 Format with no statement number
14 Function not used as variable
15 Truncated value
16 Statement out of order
17 More than 29 COMMON regions
18 Non-COMMON data

Terminating (Fatal) Errors

Terminating errors stop output of the object program. These errors are listed below.

T0	I/O error on compilation device	T5	Improper use of name
T1	Construction	T6	Improper statement number
T2	Usage	T7	Mode
T3	Data pool full	T8	Constant too large
T4	Illegal statement	T9	Improper DO nesting

Optional Listing

Source and object records can be listed. Source records are listed as they are input. Object records are listed as they are created. Each object record consists of a varying number of one to four-word object entries.

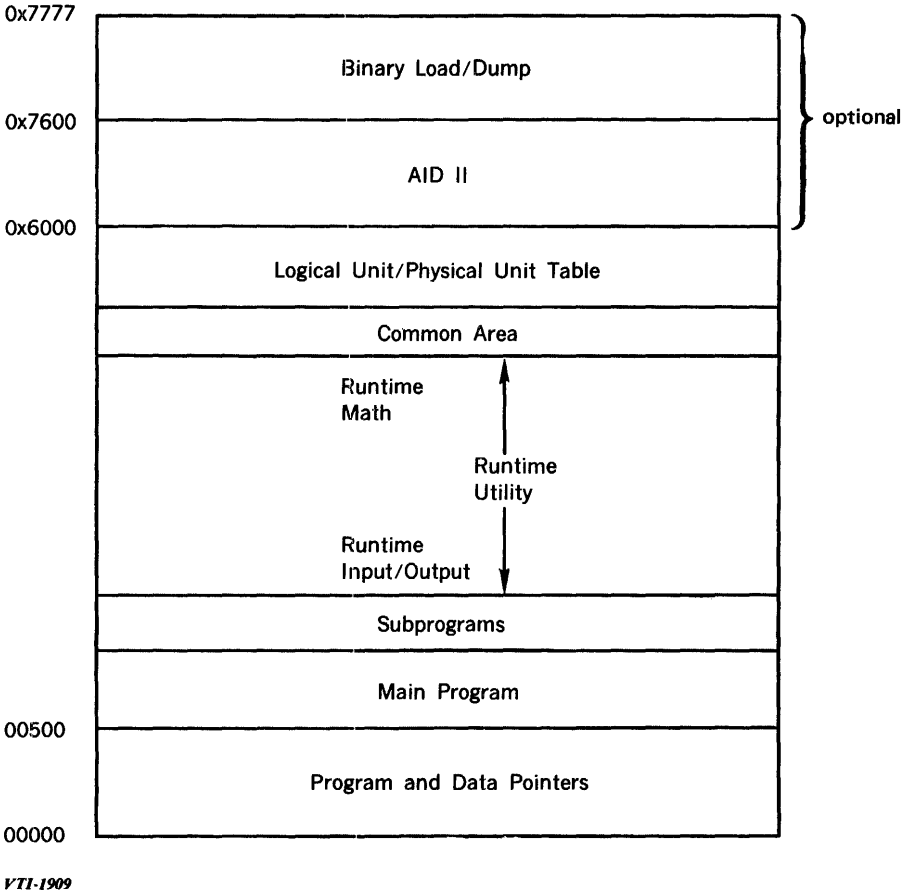
Both the MOS section of this handbook and VORTEX reference manuals contain further details of the object language. Object listings display each object word delimited by a blank as six octal digits. Each object entry occupies one record (line) on the LO unit.

Maps

The run-time memory map is shown in figure 1. In the following sample printout of a program map, the leftmost six-digit column is the value of the relative location counter of the item to the right. The letter and name, value or variable identifies the item as a relocatable (R), fixed-, or floating-point value or variable, or a subroutine entry name, the name of an External (E) subprogram or the name of a region in COMMON (C). COMMON is the name for blank or unlabeled COMMON under MOS.

Sample Program Map

```
ENTRY/COMMON BLOCK NAMES
000043 R TEST1
000004 C COMMON
000004 C      XX
EXTERNAL NAMES
000020 E     $SI
SYMBOL TABLE
000023 R 000001
000027 R 000002
000000 C       A
000002 C       B
000000 C       C
000002 C       D
000025 R       I
000005 R       5
000031 R       J
000035 R       K
000033 R 000003
000012 R      10
```



VTI-1909

Figure 1. Run-Time Memory Map

ASSEMBLING A PROGRAM

The DAS MR assembler (tape 3) is loaded and executed as follows:

a. Load the loader (tape 4) using the binary load/dump program (BLD II). Before loading, set the A register to zero to prevent execution of the loader. At completion of loading, the execution address (013260) of the loader will be in the X register.

(continued)

b. Make the following modifications to memory:

Location	New Contents
5	0210
6	0210
7	0210

To modify:

1. In STEP mode load 054000 into the instruction register.
2. Set REPEAT switch.
3. Load 000005 into P register.
4. Load 000210 into A register.
5. Press STEP or START three times (loads A into address specified by P register, which is automatically incremented by one after the instruction is executed).

c. Execute the loader by setting the P register to the execution address (013260) and pressing RUN.

d. When executed, the loader will print LN on the teletype. At this time, peripheral device assignments may be altered by entering the one-digit number of the old logical unit followed by the two-digit number of the substitute unit. DAS MR uses the following logical units:

Logical Unit Number	**Logical Unit Name**	**Default Device Assignment**
2	BO	Paper Tape Punch
3	PI	Card Reader
4	LO	Line Printer (620-77)
6	GO	Dummy
8	SS	Magnetic Tape* 00
9	PO	Magnetic Tape** 10

*Device address 010

**Device address 011

As an example of device reassignment:

LN
300400201800906

would reassign:

PI = Teletype keyboard
LO = Teletype printer
BO = Teletype paper tape punch
SS = Teletype keyboard
PO = Dummy

(continued)

Table 3 is a complete list of peripheral assignments for logical units.

Table 3. Logical Unit Assignments

Logical Unit Number	Assignment
0	Teletype keyboard and printer
1	Teletype paper tape reader and punch
2	High-speed paper tape reader/punch
3	Card reader
4	Line printer
5	Dummy
6	Dummy
7	Card punch
8	Magnetic tape unit 0
9	Magnetic tape unit 1
10	Magnetic tape unit 2
11	Magnetic tape unit 3
12	Unformatted paper tape I/O (HSPT)

e. Following device reassignments, the loader will print IN on the Teletype. At this time, the operator should ready the DAS MR object program on the input device and respond by typing the proper designation on the Teletype:

P = paper tape reader
T = Teletype paper tape reader
0,1,2,3 = magnetic tape controller 0,1,2 or 3, respectively.

To enable printing out of a load map, the operator must type M immediately following the device designator. Following the typed characters, the operator must press the carriage return key to initiate loading of the DAS MR.

(continued)

f. After DAS MR is loaded, peripheral devices for logical units 3,4,2, 6,8 and 9 must be loaded from the run-time I/O tape. This is accomplished by placing the tun-time I/O tape on the input device and repeating step e.

g. After the run-time I/O is loaded, the I/O control program must be loaded from the run-time utility tape. This is accomplished by placing the run-time utility tape on the input device and repeating step e.

h. When all externals have been satisfied, the loader will halt with the P register equal to 000003. To execute DAS MR, place the computer in the run mode.

   Upon execution, DAS MR will input source statements from logical unit 3 (PI), output source for pass 2 to logical unit 9 (PO), input pass 2 source from logical unit 8 (SS), output binary object to logical unit 2 (BO), and output listing to logical unit 4 (LO).

   Source input to DAS MR terminates upon input of either an EOF or a source record containing a slash (/) as the first character. A slash record will cause an end-of-file to be output to the BO device.

During a DAS MR assembly operation, if logical unit SS is not a magnetic tape unit, a flag bit is set in the peripheral control word PCW. When the end of pass 1 is detected, this bit is interrogated. If it is set, DAS MR does a status check on logical unit PO, prints the message RELOAD SOURCE on the Teletype, and halts. When the computer is placed in the run mode, DAS MR rewinds logical unit SS and begins pass 2 of assembly. If the flag bit is not set (SS not equal to magnetic tape), no status check is done on PO and DAS MR immediately rewinds logical unit SS and begins pass 2.

At the end of each assembly, DAS MR jumps to location 0520 to restart itself for another assembly. Therefore, assemblies may be stacked. If, at any time, a source record is input where the first character is a slash (/), DAS MR punches a trailer on logical unit BO and exits to program RSCB3 in the run-time utility (tape 6). In RSCB3, any pending I/O is executed and the computer halts with the A-register containing a current value of the error-control word (ECW). Do not place the computer in the run mode at this point. This halt is followed by a data area. To restart the assembler, go to location 0520.

Assembly Errors. The DAS MR calls EXIT when one of the following four conditions are detected on the PI device: end-of-file, beginning- or end-of-device, read error, or the first character of a record is a slash (/). When EXIT is called, the computer halts at a high memory location with the A-register indicating the following conditions:

A-Register Value	Indicated Condition
0100200	First character is a slash (/)
0100240	End-of-file detected
0100300	Read error detected
0100340	End or beginning of device detected

When the halt occurs, the error-control word (ECW) may contain more information about the problem. The address of the ECW is stored in memory location 0476. The ECW format is listed as follows:

Bit	Meaning
0-3	Loader error value: 0 = No error 1 = Checksum error 2 = Sequence error 3 = Illegal record type error 4 = Read error 5 = Illegal loader text error 6 = Data initialization error 7 = Common error 10 = Missing program(s) error 11 = Literal pool overflow error 12 = Program size error
4	0 = No loader initialization error 1 = Loader initialization error
5-7	Processor error value: 0 = No error 1 = (Not used) 2 = (Not used) 3 = Pass 2 record count not equal to pass 1 record count 4 = A system executive control directive input at an incorrect time 5 = An end of file received from I/O control 6 = An error received from I/O control 7 = An end of device or beginning of device received for I/O control No I/O control error program execution
14	0 = I/O control error program execution 1 = I/O control error program execution
15	0 = No errors in the assembled or compiled program 1 = Error in the assembled or compiled program

LOADING A PROGRAM

This section describes loading the loader (tape 4), error messages, and loading the support libraries (tapes 5 through 12).

Loading the Loader

The loader (tape 4) loads object programs in MOS format. Object-program input is from either paper or magnetic tape selected at the Teletype keyboard. Maps and error diagnostics are listed by the Teletype.

The loader is on BLD object tape and is loaded into memory by BLD II.

The loader is read initially into an area extending downward from location 015777 of memory. Upon execution, the loader performs the following operations:

a. Automatically adapts to the word size (16 or 18 bits) of the computer.

b. Relocates to upper memory of the computer in accordance with the following sense switch options:
   1. If no sense switches are set, relocates downward from location 0x5777; (x = 1, 2, 3, 4, 5, 6, 7 corresponding to memory sizes 8, 12, 16, 20, 24, 28, and 32K words, respectively).
   2. If sense switch 1 is set, relocates downward from location 0x7377.
   3. If sense switch 2 is set, relocates to occupy downward from location 0x7755.

Before loader execution begins, the following loader parameters can be either modified manually or with AID II:

Name	Location	Default value	Function
$IAP	2	010	Start of indirect address pointer table.
$LIT	5	0500	Top address +1 of literal table
$PED	7	0500	Initial loading location.

When loaded and executed, the loader types the message

```
LN
```

The operator may redefine any one or more of the first nine standard logical units (table 3) at this time by typing a string of sets of three digits for each redefinition, where:

a. first digit = the standard unit number to be changed

b. second and third digits = the substitute logical unit assigned in the range from 0 through 12 (decimal).

Only logical units 0 through 9 may be redefined. Their redefinition, however, may be any value from 0 through 12. If an error is made in any transaction, the message

```
LN
```

is retyped and the operator may repeat the entry correctly. The carriage-return key may be pressed at any time after the LN message to terminate further redefinition of logical units. Following a carriage return, the loader types the message

```
IN
```

on the Teletype. The operator should ready the object program in the proper input device and respond by typing one of the following designators depending on the desired input device:

P	High-speed paper-tape reader
T	Teletype paper-tape reader
0, 1, 2, 3	Magnetic-tape controller 0, 1, 2, or 3, respectively.

To enable printout of the loader map, the operator must type

```
M
```

immediately following the device designator or, if the loader map is to be suppressed, the operator presses the carriage-return key.

The M or carriage return following the device specification causes the loader to begin loading the binary object data from the specified input device.

If an error is detected, the loader types a 2-character error message and halts.

To continue, the operator should remove the cause of the error, ready the input device to read from the beginning of the object material, reload the loader program, and repeat the above procedure from the beginning.

A loading operation is complete when all external references are satisfied. After each normal EOF from the input device, a map of all external names is printed on the Teletype (unless suppressed as an option by the operator). If the program that corresponds to the external name has not been loaded, a

```
/
```

appears following the name in the printed map. Providing no error conditions have been detected, the loader will again type

```
IN
```

on the Teletype. The operator should follow the same procedure described above.

When the loading operation is successful, the loader prints a map of all external names (unless suppressed) and halts with

```
P = 3.
```

To execute the loaded program, the operator should press RUN.

The loader jumps to location 02 at the end of normal loading. Location 02 contains the value for loader-parameter $IAP and is always less than 01000. Therefore, the computer halts at location 03. When the computer is placed in the run mode, the computer performs a indirect jump to location 06, which contains the execution address of the program.

Error Messages

The following 2-character error messages are output to the Teletype whenever the corresponding error condition is detected:

Messages	Meaning
PS	Program Size Error. Program memory requirements exceed available program/common storage.
LS	Literal Size Error. Program literal requirements exceed available literal storage.
CM	Common Error. The program contains conflicting size definitions for a common block.
DA	Data Error. The program attempted to overlay the loader, loader tables, or resident programs.
TX	Text Error. The program object text contains an illegal or erroneous loader code.
RD	Read Error. The loader encountered a read error while attempting input of object text.
RC	Record Error. The loader inputs an invalid record type.
SQ	Sequence Error. The loader inputs an object text record with an invalid sequence number.
CK	Check-Sum Error. The loader inputs an object text record with an invalid check-sum.

Loading the Support Libraries

For most efficient use of the loader, load the support libraries in the following order:

a. Run-time I/O (tape 5)

b. Complex math functions (tape 10)

c. Double-precision math functions (tape 12)

d. Single-precision math functions (tape 11)

e. Run-time math double-precision (tape 9)

f. Run-time math single-precision (tape 7 or 8)

g. Run-time utility (tape 6)

Prepare and select the input unit for the main programs. The loader loads all required subroutines until an end-of-tape record or an end-of-file is detected, at which time the list of required subroutines is generated and input selection is again requested.

If two or more subroutines have the same name, only the first input is loaded. When all required subroutines are loaded the loader halts with the P register containing 000003. To execute the main program, place the computer in the run mode. Execution of the main program can be initiated by running at location 03.

All programs are loaded at location $PED. Required subroutines are loaded as they are input to successive blocks of memory. Common storage overlays the loader.

As in MOS, there are two 4-word buffers in the RSCB1 program of the run-time utility (tape 6). These buffers are labeled $TTL and $DAT and are used by the assembler and compiler to hold the current job name and date. They may be externally referenced by any user program.

All support libraries can be copied onto magnetic tape for loading if desired. In this case, they should all be contained in one magnetic tape file. An end-of-file mark must be written at the end of the last library copied.

PROGRAM EXECUTION AND ERROR MESSAGES

To execute a program, initialize the selected I/O devices, clear the registers, set the program counter to 000003, reset the computer and place it in the run mode.

There are three types of programmed halts: STOP, PAUSE, and CALL EXIT. STOP causes STOP to be output with the stop number, after which the computer performs a CALL EXIT.

STOP implies an end of job. PAUSE causes PAUSE to be output with the pause number, and the computer going into the step mode. The program can be continued by placing the computer in the run mode. CALL EXIT causes a HALT 07 with – 1 in the X and B registers, and signifies the end of job.

The following error halts are generated by the support libraries. These errors give a message on the Teletype. Execution will continue if the error is non-fatal; otherwise a CALL EXIT is executed.

Message	Definition	Type
FORMAT	Format error	FATAL
MODE	Data mode error (floating point versus integer)	FATAL
DATA	Input data field error	FATAL
I/O	I/O error	FATAL
GO TO RANGE	Computed GO TO out of range	FATAL
ARITH OVFL	Arithmetic overflow	NON-FATAL
FUNC ARG	Invalid function argument	NON-FATAL

The paper-tape reader driver returns reading errors to a calling program (like DAS MR) if characters not in the range 0240 to 0337 are in an ASC II record; for example, if they come after the first character in that range but before a carriage return (0215) character. Characters not in this range are allowed preceeding the record or following the carriage return.

External record formats are identical to MOS. The model-33 Teletype paper-tape punch must be turned on and off by the operator. Line printer records are 120 characters. Printing is left-justified with unused positions set to blanks.

SECTION 9 — MOS AND VORTEX OPERATING PROCEDURES

This section contains operating procedures for FORTRAN IV programming systems that are used with MOS and VORTEX.

COMPILING WITH MOS

The initiation of the MOS FORTRAN IV compiler is accomplished by entering the control directive:

/FORTRAN (or /F) P1,P2,...,Pn.

This control directive directs the executive program to call the system loader to load the FORTRAN IV compiler and commence compilation. The parameter string specifies optional tasks that are to be performed. These options are:

B No binary object program output desired.

D Integer and logical items are assigned two words.

L Load and go operation desired.

M No memory map desired.

N No source listing desired.

O Octal listing of object program desired.

X Conditional compilation desired. (Source records with an X in column 1 will be compiled.)

Input/output assignments during compilation are made through the /ASSIGN control directive (see MOS section of this handbook). The FORTRAN IV compiler uses the following logical units:

Source input	PI
Object output	BO
Listing	LO
Load and go	GO (optional)

COMPILING WITH VORTEX

The VORTEX FORTRAN IV compiler is scheduled by the job-control processor (JCP) on entry of the directive /FORT,P1,P2,...Pn, where the acceptable parameters are the same as described above for the MOS directive. The logical units used are the same as those for MOS, and assignments are made using the JCP directives /ASSIGN and /PFILE.

LOADING WITH MOS

To run a program compiled under MOS FORTRAN IV, initialize the MOS, and load the compiled object program with the MOS directive:

/LOAD (or /L)

The error messages are the same as in the stand-alone version (section 8) and are listed as follows:

Message	Meaning
PS	Program Size Error. Program memory requirements exceed available program/common storage.
LS	Literal Size Error. Program literal requirements exceed available literal storage.
CM	Common Error. The program contains conflicting size definitions for a common block.
DA	Data Error. The program attempted to overlay the loader, loader tables, or resident programs.
TX	Text Error. The program object text contains an illegal or erroneous loader code.
RD	Read Error. The loader encountered a read error while attempting input of object text.
RC	Record Error. The loader inputs an invalid record type.
SQ	Sequence Error. The loader inputs an object text record with an invalid sequence number.
CK	Check-Sum Error. The loader inputs an object text record with an invalid check-sum.

LOADING WITH VORTEX

Run-time error messages are the same as those listed above for MOS loading.

Non-Resident Programs

The object program output by the VORTEX compiler is input to the load module generator (LMGEN). The job-control processor schedules LMGEN upon inputting the directive: /LMGEN. LMGEN creates a load module on the system-workfile SW device on inputting the following four directives:

TIDB,name, bf,s,DEBUG	bf = 1 for background; 2 for foreground s = Overlay count DEBUG is optional and loads the DEBUG routine when present
LD,obj	obj = specifier giving object module logical unit number and key, lun/key
LIB,lib	lib = specifier giving library lun/key
END,save	save is optional for specifying the load module save lun/key

The program can then be loaded and executed from SW by entering /EXEC on the System Input (SI) device; or, if bf = 1 and save = BL,E, it can be executed by the JCP directive /LOAD, name; or if bf = 2, it can be scheduled by entering the OPCOM directive ;SCHED,name,level,save, or by another task, using the SCHED macro.

Resident Programs

The object program output by the VORTEX compiler is input to the SGEN program and made part of the VORTEX nucleus (see VORTEX reference manual 98 A 9952 101). All required subroutines must be added at this time.

I/O DEVICE CONTROL

The I/O control components of MOS and VORTEX permit access to I/O devices through the use of logical units. A logical unit is an I/O device or partition of a rotating-memory

device (RMD). A program references an assigned number. The logical unit numbers permit I/O operations independent of the phsyical-device configuration. For further information on logical units, refer to the input/output control description in the MOS section of this handbook or the VORTEX reference manual.

The FORTRAN IV compiler inputs source text from logical unit PI, outputs listings and maps on logical unit LO, and produces an object module (code and loading information) on logical units BO and GO. For further information, refer to the FORTRAN IV compiler description in the MOS section of the handbook or the VORTEX reference manual.

COMPILER INPUT RECORDS WITH MOS

The compiler requests 40-word (80-character) input records from the Input/Output Control System (IOCS). For further information on IOCS, refer to the MOS section of this handbook.

COMPILER INPUT RECORDS WITH VORTEX

The compiler requests 40-word (80-character) input records from IOCS, if PI is not a rotating memory device (RMD) or if PI = SI. Otherwise, the compiler inputs 120-word records (three FORTRAN source records) from the RMD, and does its own deblocking. FORTRAN RMD source modules must start on a record boundary.

COMPILER OUTPUT RECORDS WITH MOS

The compiler outputs 60-word (120-character) records to logical unit LO. An object module produced on logical units BO and GO is in 60-word records for Varian 16-bit computers and 53-word records for the 18-bit computers. The MOS section of this handbook describes the object module format.

COMPILER OUTPUT RECORDS WITH VORTEX

Output records are 60 words long. An object module produced on an RMD is blocked two records for each RMD record. FORTRAN object modules start on the RMD-record boundary. The VORTEX reference manual (appendix A) describes the object module format.

ERROR MESSAGES

Error messages (notification and terminating) occurring under MOS and VORTEX are the same as for the stand-alone version (section 8). For further information on error messages, refer to the FORTRAN IV compiler descriptions in the MOS section of this handbook or the VORTEX reference manual.

MAPS WITH MOS

Illustrations of a run-time memory map are provided in the description of the system loader control directives in the MOS section of this handbook.

Program maps are the same as in the stand-alone version (section 8).

MAPS WITH VORTEX

The FORTRAN IV compiler is a level 1 background program in the VORTEX system. Memory maps are generated by LMGEN and SYSGEN directives, and are the same as in the stand-alone version. Program maps are generated in the same manner as in the stand-alone version (section 8).

BASIC Language

TABLE OF CONTENTS

SECTION 1
A PRIMER IN BASIC

SECTION 2
ADVANCED BASIC

SECTION 3
STATEMENTS IN BASIC

SECTION 4
USING THE BASIC SYSTEM

SECTION 5
ERROR MESSAGES

SECTION 6
CALL STATEMENT DESIGN CONSIDERATIONS

SECTION 7
EXTENDED BASIC

NOTES

In the examples given in this manual, **boldface** type indicates obligatory items, and *italics* indicate optional items. Capital letters indicate precisely the letters used, and small letters indicate that other letters and or numbers are to be substituted.

SECTION 1 — A PRIMER IN BASIC

A **program** is simply a set of directions that tells a computer how to solve a problem. The computer takes the raw-data input, manipulates it according to the directions in the program, and, if there are no errors in the data or program, gives the answer required.

For proper performance, any program must fulfill two requirements:

a. *The program must be in a language that is understood by the computer*, just as problems presented to people must be in languages they understand.

b. *The program must be complete and precise* because the computer, unlike human problem-solvers, cannot make inferences. The computer does what you order, not what you meant to order.

A program in English would pose insurmountable difficulties for the computer. English and other human languages are rich in ambiguities and redundancies, qualities that make poetry possible and computing impossible. Thus, you must present your program in a language that has many of the characteristics of ordinary mathematical notation: simple vocabulary and grammar, but with the ability to specify problem-solving steps completely and precisely. One such language is the **B***eginner's* **A***ll-purpose* **S***ymbolic* **I***nstruction* **C***ode* (**BASIC**), originally developed at Dartmouth College.

In this manual, the rest of section 1 introduces you to the BASIC language and shows you how to write simple programs that can solve a wide variety of useful and interesting problems. Section 2 shows you how to apply the BASIC language to more advanced computer techniques. Section 3 shows you how to use and operate the BASIC system, and includes a variety of reference material.

AN EXAMPLE

Let us begin by seeing how the BASIC language is applied to the solving of a system of two simultaneous liner equations in two variables, and then to the solving of two different systems, each differing from the first only in the constants c and f. Given:

$$ax + by = c$$

$$dx + ey = f$$

Then

$$x = (ce - bf)/(ae - bd)$$

$$y = (af - cd)/(ae - bd)$$

Note that, if ae − bd = 0, there is either no solution or an infinite number of solutions, but no solution that is unique. In any other case, there will be a unique solution.

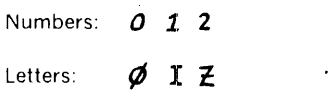
Whether or not you understand the manual solution of such systems is not important. For now, study the following example and explanation to learn how a BASIC program for solving the problem is developed.

EXAMPLE

```
10    READ A, B, D, E
15    LET G = A * E - B * D
20    IF G = 0 THEN 65
30    READ C, F
37    LET X = (C * E - B * F)/G
42    LET Y = (A * F - C * D)/G
55    PRINT X, Y
60    GO TO 30
65    PRINT "NO UNIQUE SOLUTION"
70    DATA 1, 2, 4
80    DATA 2, -7, 5
85    DATA 1, 3, 4, -7
90    END
```

Several things about the program can be noted:

a. *It uses only capital letters* since the Teletype has only capital letters. A handwritten program separates confusing pairs of characters by adhering to the following conventions:



Since these characters are on different Teletype keys, there is no confusion during typing.

b. *Each line of the program begins with a number* called a **line number** that identifies the line, called a **statement**, and specifies the order in which the statement is to be processed by the computer. The program can be written in any order since the computer will sort it and edit it as specified by the line numbers.

c. *Each statement has a word following the line number.* This word specifies the **type of statement**.

d. *Each statement is free form.* This is not obvious from the program printed above, but spaces have no effect on statements in BASIC. Thus, line 10 could have been typed as

```
10READA,B,D,E
```

and line 15 as

```
15LETG=A*E-B*D
```

The exception to this is the spacing in statements to be printed (e.g., line 65) since they print just as written, including spacing.

Now let us go through the example, statement by statement.

The first statement, line 10, is a READ statement. READ statements must be accompanied by one or more DATA statements, which do not, as the example shows, have to be adjacent to the READ statement in the program. Whenever the computer encounters a READ statement while executing a program, it assigns the next available values in the DATA statement(s) to the variables in the READ statement. Thus, the variable A in line 10 is assigned the value 1 from line 70, the first DATA statement. Similarly, B is assigned the value 2, D the value 4, and E the value next available in the DATA statements, i.e., the value 2 from line 80. The next READ statement will pick up DATA values from where this statement leaves off. This is further discussed below.

The second statement, line 15, is a LET statement. LET statements contain formulas to be evaluated using mathematical notation slightly modified to meet the requirements of the computer, e.g., use of the asterisk (*), which cannot be omitted, to denote multiplication. In line 15, we order the computation of AE − BD and call the result G. In general, a LET statement directs the computer to set a single variable equal to a mathematical expression, where the variable is to the left of an equal sign and the mathematical expression to the right. *Note that in a LET statement the equal sign does not denote equality but replacement, and line 15 might best be read as " replace the quantity AE − BD with G" ; thus, formulas such as X = X + 1 are perfectly valid in LET statements.*

The next statement, line 20, is an IF/THEN statement. IF/THEN statements contain conditions that, if met, cause the execution of the program to go next to the statement designated after the THEN. In this example, we know that there can be no unique solution if G = 0, so we order the program to jump to line 65 if, and only if, G = 0.

The statement in line 65 is a PRINT statement. PRINT statements cause the output of the material between quotation marks and/or of computed values. Thus, if G = 0 (line 20), the program prints the text

```
NO UNIQUE SOLUTION
```

and, since lines 70, 80, and 85 contain nonexecutable DATA statements, passes to line 90, an END statement. This terminates the execution of the program for the case where G = 0.

However, in our example, $G \neq 0$. Since the condition in line 20 is not met, the program continues to execute statements in the order of their appearance, rather than jumping to line 65. Thus, an IF/THEN statement tells the computer where to go for its next instruction when the IF condition is met, but, if the condition is not met, the computer passes to the next statement in sequence.

The next statement, line 30, is another READ statement that assigns the next two available DATA values, −7 and 5 (both from line 80), to the variables C and F, respectively. This supplies the rest of the required constants to the original pair of equations to yield the system

$$x + 2y = -7$$

$$4x + 2y = 5$$

The next statements, lines 37 and 42, are LET statements that direct the computer to find the values of X and Y according to the formulas provided. Note the use of parentheses to indicate that CE − BF is to be divided by G. Had the parentheses been omitted, the computer would have divided only BF by G, solving

$$x = ce - (bf/g)$$

rather than the required

$$x = (ce - bf)/g$$

that is equivalent to the original

$$x = (ce - bf)/(ae - bd)$$

The next statement, line 55, is a PRINT statement that causes the output of the computed values of X and Y. The values 4 and 5.5 are printed and the computer goes to the next statement.

The next statement, line 60, is a GO TO statement. A GO TO statement causes the execution of the program to go next to the statement designated. There is no condition. When the computer encounters a GO TO statement, it *always* goes to the statement designated.

The computer thus returns to line 30, which contains a READ statement. The next two available DATA values, 1 and 3 (from line 85), are assigned to the variables C and F, respectively. The old values of C and F are lost and replaced with the new values. This yields the system

$$x + 2y = 1$$

$$4x + 2y = 3$$

As before, the computer finds the values according to lines 37 and 42, prints the results as directed in line 55, and returns to line 30 as specified by line 60.

Here the next two values of C and F, 4 and −7, respectively (from line 85), are assigned and the computer solves the system

$$x + 2y = 4$$

$$4x + 2y = -7$$

It prints the solutions and returns to line 30. However, there are no more DATA values for assignment to C and F. The computer informs us that it is out of data by printing on the Teletype

```
ERROR 56 IN LINE 30
```

(Error 56 does not indicate a *mistake* in the program, but merely conveys information from the computer concerning the lack of data. We will examine such **error messages** in detail later.)

This terminates the execution of the program. Note that it is not necessary to reach an END statement to terminate.

The program and the results are shown below just as they would appear on the Teletype. After typing in the program, type the word RUN and press the RETURN key. This directs the computer to execute the program.

Program

```
10    READ A, B, D, E
15    LET G = A*E - B*D
20    IF G = 0 THEN 65
30    READ C, F
37    LET X = (C*E - B*F)/G
42    LET Y = (A*F - C*D)/G
55    PRINT X, Y
60    GO TO 30
65    PRINT "NO UNIQUE SOLUTION"
70    DATA 1, 2, 4
80    DATA 2, -7, 5
85    DATA 1, 3, .4, -7
90    END
RUN
```

Result

```
4                    -5.5
0.666666             0.166666
-3.66666             3.83333
ERROR 56 IN LINE 30
```

Now that we have solved the problem, let us examine more closely certain characteristics of the program.

For example, omission of line 20 would not have affected the solution just presented. However, in the case where ae − bd = 0, omission of line 20 would have required the computer to perform the impossible calculation of dividing by zero in lines 37 and 42. The computer would then merely print

```
ERROR 69 IN LINE 37

ERROR 69 IN LINE 42
```

where ERROR 69 indicates an attempt to divide by zero.

Omission of line 55 would have been catastrophic. The computer would have made the required calculations, but could not print them since it was not ordered to do so. The solutions would have remained the computer's secret. Furthermore, there would have been no error message to signal that something was amiss, since there was no error in the format of the program, the data, or the computations. The Teletype would have simply remained blank.

Omission of line 60 would not have affected the first set of solutions, but after the initial values of X and Y were printed, the computer would have gone to line 65, printed

```
NO UNIQUE SOLUTION
```

and stopped. At least the curious juxtaposition of a set of solutions followed by this message would have brought the error to our attention.

The choice of individual line numbers is arbitrary, but the lines must be numbered in the order to be followed during the execution of the program (disregarding jumps caused by GO TO and IF THEN statements). We could have numbered the lines 1, 2, 3, ... 13, although this is not recommended. Spacing between the numbers allows for insertions made necessary by omissions or by modifications. Thus, if we discover that we have left out two statements between lines 40 and 50, we can assign them numbers such as 44 and 46 so that the computer will place them in the proper order during the editing and sorting of the program.

The division of data items among the DATA statements is arbitrary, but the items must appear in the order in which they are to be read during the execution of the program. Thus, in our example, the first data item is assigned to the variable A, the second to B, the third to D, the fourth to E, the fifth to C, the sixth to F, the seventh to C (replacing the fifth), etc. Rather than the three lines 70, 80, and 85 given in the example, we might have written

```
75   DATA 1, 2, 4, 2, -7, 5, 1, 3, 4, -7
```

or perhaps more naturally

```
70    DATA 1, 2, 4, 2
75    DATA -7, 5
80    DATA 1, 3
85    DATA 4, -7
```

to indicate that the coefficients appear in the first DATA statement and the various pairs of values for C and F in the subsequent DATA statements.

FORMULAS

The computer computes by evaluating the formulas in a program. These formulas are similar to those used in standard mathematical notation, with slight modifications required by the nature of the computer.

One general limitation in BASIC is that a formula must be written on a single line of the coding sheet or punched in a single card. However, long formulas can be broken down into two or more short formulas that meet this requirement. The limitation is thus one of writing and not one of computation.

Operations

BASIC uses the following five *arithmetic operations* in formulas, each indicated by the corresponding symbol. Note that the asterisk (*) used to indicate multiplication cannot be omitted, i.e., in ordinary mathematical notation AB is equivalent to A x B or A · B, but in BASIC this can be expressed only by A * B.

Operation	Symbol	Example	
Addition	+	A + B	(Add B to A)
Subtraction	-	A - B	(Subtract B from A)
Multiplication	*	A * B	(Multiply A by B)
Division	/	A / B	(Divide A by B)
Exponentiation	↑	A ↑ B	(Raise A to the power of B)

BASIC also provides for evaluation of the following six *arithmetic relationships* in IF/THEN statements where such comparisons define the IF condition.

Relationship	Symbol	Example	
Is equal to	=	A = B	(A is equal to B)
Is not equal to	#	A # B	(A is not equal to B)
Is less than	<	A < B	(A is less than B)
Is greater than	>	A > B	(A is greater than B)
Is not less than	> =	A > = B	(A is equal to or greater than B)
Is not greater than	< =	A < = B	(A is equal to or less than B)

The introductory example contained an instance of the use of an arithmetic relationship in the statement

```
20 IF G = 0 THEN 65
```

In addition to the arithmetic operations and relationships, the computer can evaluate several *mathematical functions* in BASIC:

Function		Coding
Find the sine of x	Where x is a number or is an angle measured in radians	SIN (X)
Find the cosine of x		COS (X)
Find the tangent of x		TAN (X)
Find the arctangent of x		ATN (X)
Find e to the power of x		EXP (X)
Find the natural logarithm of x (ln x)		LOG (X)
Find the absolute value of x (| x |)		ABS (X)
Find the square root of x ($\sqrt{x}$)		SQR (X)
Truncate x*		INT (X)

Function	Coding
Randomize x*	RND (X)
Assign a sign to x*	SGN (X)

* These functions, which are not self-explanatory, are discussed in section 2.

In coding the above functions, we can substitute any number or mathematical expression for X. For example, to find the square root of (4 + x), we would write

```
SQR (4+X↑3)
```

Parentheses

To ensure that items in formulas are properly grouped together for correct execution of a program, BASIC requires careful use of parentheses. Such use gives attention to the order in which the computer performs the calculations.

An expression inside parentheses is computed before the quantity is used in further computation. For example, in the expression A −(B + C), the quantity B + C is first computed, and then the result subtracted from A. In the case of nested parentheses, the nests are computed from the inside out, e.g., in A −(B −(C + D)), the quantity C + D is computed first.

In the absence of parentheses in an expression or part of an expression, the computer performs its calculations according to the following three levels of priority:

a. First, all exponentiation

b. Next, multiplication and division

c. Finally, addition and subtraction

Within a level of priority, computations are performed from left to right. For example, if we type A + B * C ↑ D, the computer first raises C to the power D, multiplies the result by B, and then adds A to the product. If this is not the order intended, we specify the desired order by the use of parentheses: to raise the product of B and C to the power D before addition to A, we write A + (B * C) ↑ D; to multiply A + B by C and raise the product to the power D, we write ((A + B) * C) ↑ D.

If there is any question in your mind about these priorities, add more parentheses to eliminate possible ambiguities. Thus, the computer, faced with A − B − C, first subtracts B from A and then subtracts C from their difference. Faced with A / B / C, it first divides A by B, then divides that quotient by C. Given A ↑ B ↑ C, the computer raises A to the power B, then raises that result to the power C. Changing or clarifying this order of computation requires the use of parentheses.

Parentheses permit easy formulation of complicated mathematical expressions. Thus, to find the arctangent of the quantity

$$3x - 2e^n + 8$$

we write

```
ATN (3 * x - 2 * EXP (N) + 8)
```

Or, to find the value of $(5/8)^{17}$, simply write the two-line BASIC program

```
10 PRINT (5/8) ↑ 17
20 END
```

and the computer calculates the decimal form of the answer and prints it in less time than it took to type the program.

Numbers

In BASIC, a **number** is a positive or negative decimal value of up to (approximately) seven significant digits. The following are thus valid numbers in BASIC: 2, -3.675, 1234567, -.7654321, and 483.4156.

The following are *not* valid numbers in BASIC: 14/3 and $\sqrt{7}$. These are expressions, not numbers, and must be converted by the computer to valid BASIC numbers before further manipulation or inclusion in a list of data. In the first case, the expression contains two numbers, 14 and 3, whose quotient in decimal form can be manipulated or included in a list of data. The second expression contains one number, 7, whose square root can be computed in BASIC by applying the appropriate mathematical function, SQR (7), to yield a valid decimal number that can be used in further computations or in a list of data.

Additional range and flexibility in the BASIC number system is attained by using the letter E (exponent), which is read " times ten to the power," e.g., 73E7 indicates 73 times 10 to the power 7. Thus, we can write 0.001234567 in several forms acceptable in BASIC, e.g., .1234567E -2, 1234567E -9, 1234.567E -6. We can write ten million as 1E7 or 1E+7, but we cannot write just E7 (this is a variable, section 1.2.4) since we must indicate 1 as the quantity that is to be multiplied by 10

Variables

In BASIC, a variable is denoted by a single letter or by a single letter followed by a single digit. The following are thus valid variables in BASIC: E7, A, X, X9, and Q2.

A variable in BASIC stands for a number, usually one whose value is not known when the

program is being written. Variables are assigned numerical values by LET, READ, and INPUT statements. Values thus assigned do not change until the next such statement containing a new value for that variable is encountered. Then, the new value replaces the former value and is used in subsequent manipulations.

A numerical value must be assigned to a variable before it can be used in a computation since all variables are undefined before each run. Failure to make such assignments results in the error message

```
ERROR 50 IN LINE nn
```

LOOPS

Often programs contain portions to be performed repeatedly, sometimes with slight changes on each pass. Iterations and incrementations are examples of such cases. In BASIC, the **loop** simplifies the writing of such programs because the portion to be repeated is written only once.

For example, a BASIC program to print a table of the first 100 positive integers and their square roots would be 101 lines long without the use of loops:

```
10    PRINT 1, SQR (1)
20    PRINT 2, SQR (2)
30    PRINT 3, SQR (3)
      .
      .
      .
990   PRINT 99, SQR (99)
1000  PRINT 100, SQR (100)
1010  END
```

However, the use of a loop reduces the 101-line program to five lines giving the same results:

```
10    LET X = 1
20    PRINT X, SQR (X)
30    LET X = X + 1
40    IF X <= 100 THEN 20
50    END
```

Line 10 assigns X the value of 1. Line 20 prints this value of X and its square root. Line 30 increases the value of X to 2 (the statement is read " let the new value of X be the old value of X plus one"). Line 40 asks if the new value of X is less than or equal to 100, and, since this is true, directs the computer back to line 20. Line 20 prints 2 and the decimal form of $\sqrt{2}$. Line 30 then increases the value of X to 3, line 40 returns the computer to line 20, etc., repeating the loop in this manner 100 times. On the 101st pass, however, the relationship in line 40 is false since X is now 101. Therefore, the computer does not return to line 20, but goes to line 50 and ends the program. This example shows the *four*

characteristics of a loop: *initialization* (line 10), *body* (line 20), *modification* (line 30), and *exit test* (line 40).

Because the necessity for the use of loops of the type just illustrated arises so often, BASIC provides FOR and NEXT statements to specify such a loop even more simply. Our sample program then reduces to

```
10    FOR X = 1 TO 100
20    PRINT X, SQR (X)
30    NEXT X
50    END
```

Line 10 specifies a range of values, and line 30 increments X and returns the computer to line 20. When the range specified in line 20 is exhausted, there is no " next X" and the computer goes to line 50 to terminate the program.

The above illustration shows the simplest form of the FOR statement where the variable is incremented by one on each pass. However, other increments can be specified by a STEP clause in the FOR statement. Thus, the above example can be modified to increment in steps of 5 by writing

```
10    FOR X = 1 TO 100 STEP 5
```

where the computer would assign 1 to X on the first pass, 6 on the second, 11 on the third, etc., up to 96. Since the next increment would be 101, which is out of the specified range, the program terminates after printing 96 and its square root.

STEP can be negative. The above example can be modified to print the table of square roots in reverse order by writing

```
10    FOR X = 100 TO 1 STEP -1
```

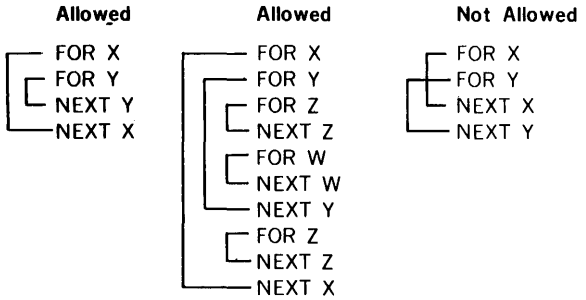
FOR statements can contain initial values, final values, and step sizes that are expressions of any required complexity. Thus, provided N and Z have been specified earlier in the program, we can write such statements as

```
99    FOR A = N + 7 * Z TO (Z - N)/3 STEP (N - 4 * Z)/10
```

The loop continues as long as the value of the control variable (i.e., the variable to the left of the equal sign) is within the specified range. This range can be specified by the conditions of an IF statement or by a FOR statement. If the initial value specified is already outside the range given in the IF statement, the body of the loop is not performed. The computer goes immediately to the line following the NEXT statement. Thus, in the following program for adding up the first n integers will give the correct result of zero when N is zero.

```
10    READ N
20    LET S = 0
30    FOR K = 1 TO N
40    LET S = S + K
50    NEXT K
60    PRINT S
70    GO TO 10
90    DATA 3, 10, 0
99    END
```

Loops within loops are called **nested loops** and can be programmed with FOR and NEXT statements. However, nested loops must actually nest since loops that cross are invalid, as shown below.



ARRAYS

BASIC can generate arrays by the use of **array variables** that consist of a single letter and are followed by subscripts in parentheses. The subscripts can be single, for example, to specify the coefficients of a polynomial (a_1, a_2, a_3, ...), or double, as in a two-dimensional matrix ($b_{n,m}$). In BASIC, the first is written A(1), A(2), A(3), ... and the second B(1,1), B(1,2), ...

The letter used for an array variable can also be used as an ordinary BASIC variable (section 1.2.4) without confusion. However, within the same program, a letter cannot be used for both singly and doubly subscripted array variables.

The form of the subscript is flexible, and can be a number, variable, array variable, or mathematical expression. Thus, BASIC permits array elements such as B(I,K) and Q(A(3,7),B - C).

We can enter the one-dimensional array A(1), A(2), ..., A(10) into a program very simply:

```
10    FOR I = 1 TO 10
20    READ A(I)
30    NEXT I
```

```
40    DATA 2, 3, -5, 5, 2.2, 4, -9, 123, 4, -4
```

This simple form of array specification is all that is required for singly subscripted arrays in which no subscript is greater than ten. To specify larger arrays, use a DIM (dimension) statement of the form

```
DIM x(n)
```

where x is the array variable and n is the highest subscript in the array x. The DIM statement tells the computer to save sufficient space for the array. It is therefore advisable to allow for the maximum possible number of entries when the size of the array is not precisely known. For example, to enter a list of 15 numbers, we might write

```
10    DIM A(25)
20    READ N
30    FOR I = 1 TO N
40    READ A(I)
50    NEXT I
60    DATA 15
70    DATA 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
80    DATA 37, 41, 43, 47
```

Note that lines 20 and 60 could have been eliminated by writing FOR I = 1 TO 15, but the longer form given allows for lengthening the array up to 25 elements by changing only line 60. Of course, to extend the array beyond 25 elements requires, in addition, changing line 10.

Similarly, we can write the two-dimensional array B(1,1), B(1,2), ..., B(3,5) into a program as:

```
10    FOR I = 1 TO 3
20    FOR J = 1 TO 5
30    READ B (I,J)
40    NEXT J
50    NEXT I
60    DATA 2, 3, -5, -9, 2
70    DATA 4, -7, 3, 4, -2
80    DATA 3, -3, 5, 7, 8
```

No DIM statement is required as long as no subscript is greater than 10, i.e., the above entry could be expanded to include all entries up to B(10,10) without a DIM statement An attempt to write an array with a subscript product greater than 100 without using a DIM statement yields the error message

```
ERROR 49 IN LINE xx
```

The error can be corrected by entering the missing DIM statement. To specify a 20-by-30

table in the above program, write

```
5     DIM B(20,30)
```

Since a DIM statement is merely a specification and is not executed during the program, it can be entered on any line before the END statement. However, it is convenient to place DIM statements near the beginning of the program.

The DIM statement is usually used to save more space than the ten subscripts automatically allowed by the computer, but in the special case of a long program containing many short arrays, DIM can be used to allot less space to arrays in order to leave more for the program.

Below is a program using both a singly and a doubly subscripted array. It computes the total sales of each of five salesmen, all of whom sell the same three products. Array P, specified in lines 10 through 30, gives the price-per-unit of each product. Array S, specified in lines 40 through 80, tells how many units of each product were sold by each salesman. The program reads the price data from line 900 into the elements of array P, and the sales data from lines 910 through 930 into the elements of array S. Thus, product 1 sells for $1.25 per unit, product 2 for $4.30, and product 3 for $2.50; salesman 1 sold 40 units of the first product, 10 of the second, etc. To enter the sales for the next month and reuse the program for those data requires changing only lines 910 through 930. A price change requires a change to line 900.

Program

```
      READY
10    FOR I = 1 TO 3
20    READ P(I)
30    NEXT I
40    FOR I = 1 TO 3
50    FOR J = 1 TO 5
60    READ S(I,J)
70    NEXT J
80    NEXT I
90    FOR J = 1 TO 5
100   LET S = 0
110   FOR I = 1 TO 3
120   LET S = S+P(I)*S(I,J)
130   NEXT I
140   PRINT "TOTAL SALES FOR SALESMAN" J; "$" S
150   NEXT J
900   DATA 1.25, 4.30, 2.5
910   DATA 40, 20, 37, 29, 42
920   DATA 10, 16, 3, 21, 8
930   DATA 35, 47, 29, 16, 33
999   END
RUN
```

Result

```
TOTAL SALES FOR SALESMAN 1 $180.5
TOTAL SALES FOR SALESMAN 2 $211.3
TOTAL SALES FOR SALESMAN 3 $131.65
TOTAL SALES FOR SALESMAN 4 $166.55
TOTAL SALES FOR SALESMAN 5 $169.4
```

ERRORS AND DEBUGGING

More often than not, the first running of a program will reveal **errors**. Program errors are of two types:

a. A **format (grammatical) error** produces an error message (section 5) that informs you of the type and location of the mistake so that you can correct it easily.

b. A **logical error** does not produce an error message since it does not have any characteristics that appear to the computer as mistakes. A logical error can, however, result in incorrect results, or no results at all. The logical error is thus more difficult to detect and correct than the format error, particularly when the results seem somewhat reasonable.

In either case, the error is first isolated and then it is corrected by inserting, deleting, or changing lines in the program.

a. *To insert a line in a program*, type the new line using a line number between the line numbers of the lines that are to precede and follow the new line; e.g., to insert a line between lines 70 and 80, type the new line with a line number of 75.

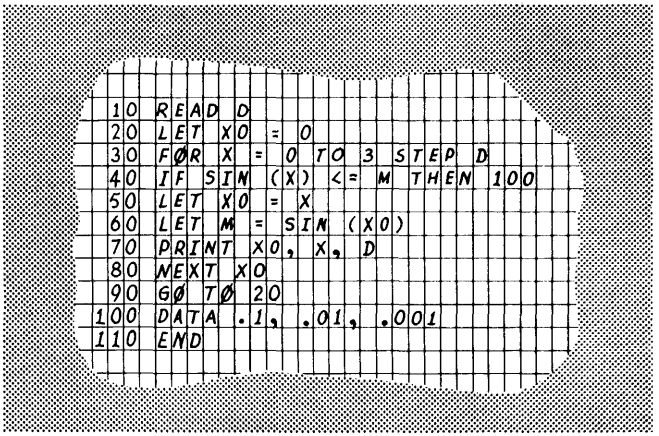
b. *To delete a line from a program*, type only the line number and then press RETURN.

c. *To change a line*, retype the line correctly using the same line number.

Such corrections can be made at any time because the computer sorts the line numbers automatically.

The rest of this section is an illustration of the process of debugging (correcting) a program based on the problem of finding that value of x between 0 and 3 for which the sine of x is a maximum, and ask the machine to print out this value of x and the value of its sine. If you have studied trigonometry, you know that $\pi/2$ is the correct value, but we shall use the computer to test successive values of x from 0 to 3, first using intervals of 0.1, then of 0.01, and finally of 0.001. Thus, we ask the computer to find the sine of 0, of 0.1, of 0.2, of 0.3, ..., of 2.8, of 2.9, and of 3, and to determine which of these 31 values is the largest. The computer does this by testing SIN(0) and SIN(0.1) to determine which is

larger, and calls the larger of these two numbers M. It then picks the larger of M and SIN(0.2) and this value is called M. This number is checked against SIN(0.3), etc. Each time a larger value of M is found, the value of x is remembered in X0. When the computer completes this process, M will have been assigned to the largest value. The search is repeated; this time the computer checks the 31 numbers (0, 0.01, 0.02, 0.03, ..., 2.98, 2.99, and 3), finding the sine of each and determining which is the largest. After this, the computer makes a third run using increments of 0.001. At the end of these three runs, the computer is to print three values: (1) the value of X0 that has the largest sine, (2) the sine of that value, and (3) the interval used on the run that found the value. To solve this problem, we begin by writing the program:



Now we are ready to enter the program on the Teletype. The sequence shown below is the entire sequence that appears on the Teletype, followed by explanatory comments.

```
READY
10    READ D
20    LWR X0 = 0
ERROR 11 IN LINE 20

20    LET X0 = 0
30    FOR X = 0 TO 3 STEP D
40    IF SINE - (X) <= M THEN 100
50    LET X0 = X
60    LET M = SIN (X)
70    PRINT XO, X, D
ERROR 36 IN LINE 70

70    PRINT X0, X, D
80    NEXT Z-X0
90    GO TO 20
```

```
100   DATA .1, .01, .001
110   END
RUN
ERROR 40 IN LINE 30
```

Comments: *Line 20:* The error message indicates that LET is mistyped. The correction is made by retyping the line correctly.

Line 40: The letter E is incorrectly entered after SIN. Typing a backarrow deletes the character immediately preceding. When an error is caught early enough to use this simple corrective technique, it is employed. The incorrect character is then replaced by the correct one (in this case, a blank) by typing it after the backarrow.

Line 70: The second error message indicates that XO is used for a variable rather than the correct form X0. The correction is made by retyping the line correctly.

Line 80: Another use of the backarrow replaces the incorrect character Z with the correct character X.

Last line: After typing the END statement, we try to run the program by typing RUN. However, this causes another error message, this time to advise us that the program contains a FOR statement without a corresponding NEXT. Upon checking, we see that this is caused by having the variables in lines 40 and 80 different. This is corrected by retyping line 80 using the same variable as that found in line 40. Furthermore, the IF/THEN statement in line 40 directs the computer to a DATA statement instead of to line 80. Line 40 is thus also retyped and another attempt made to run the program.

```
80    NEXT X
40    IF SIN (X) <= M THEN 80
RUN
ERROR 50 IN LINE 40
```

The new error message indicates that M has never been assigned an initial value. (This error came to the attention of the computer only after line 40 had been corrected.) We assign M an initial value of − 1 and make another attempt to run the program.

```
20    LET M = -1
RUN
```

Result

```
0      0      .1
.1     .1     .1
.2     .2     .1
.3     .3     .1
.4     .4     .1
READY
```

We are now getting results, but they are incorrect. We are receiving every value of X, X0, and interval size. Therefore, the printout is stopped by typing any character on the Teletype while it is running. This error is corrected by typing:

```
70
85 PRINT X0, M, D
RUN
```

to move the PRINT statement outside the loop. Typing line number 70 followed by a carriage return deletes that line. It is retyped using line number 85 and with the incorrect variable X replaced by the correct variable M.

This attempt to run the program yields the following results:

```
1.59999     .999573     .1
1.59999     .999573     .1
1.59999     .999573     .1
1.59999     .999573     .1
READY
```

because line 90 returns the computer to line 20, merely repeating the operation using the same values, rather than to line 10 to pick up a new value for D. At the same time that we make this correction, we decide to add headings for the columns of figures of the results. Thus, we type:

```
90    GO TO 10
5     PRINT "X VALUE", "SIN", RESOLUTION"
ERROR 21 IN LINE 5
```

The new error message indicates the format error in line 5, in which there is no left quotation mark for the third item. We retype line 5 correctly and run the program.

```
5     PRINT "X VALUE", "SIN", "RESOLUTION"
RUN
```

Result

```
 X VALUE     SIN          RESOLUTION
1.59999     .999573      .1
1.56998     .999999      1.00000E-02
1.5709      .999999      1.00000E-03
ERROR 56 IN LINE 10
```

Thus, we now obtain the correct results as specified in the original problem. (Remember that ERROR 56 does not indicate a mistake, but merely that there are no more data.) Having changed so many parts of the program, we request a list of the corrected program. Periodic listing of the present state of the program is an important debugging aid. The listing is requested merely by typing LIST. Typing PLIST at the end of the listing causes

the program to be punched on paper tape for later use. The correct listing appears on the Teletype as:

```
LIST
5     PRINT "X VALUE", "SIN", "RESOLUTION"
10    READ D
20    LET M = -1
30    FOR X = 0 TO 3 STEP D
40    IF SIN (X) <= M THEN 80
50    LET X0 = X
60    LET M = SIN (X)
80    NEXT X
85    PRINT X0, M, D
90    GO TO 10
100   DATA .1, 1.00000E-02, 1.00000E-03
110   END
PLIST
```

One common debugging aid that we have not used is the insertion of a PRINT statement to check that the computer is doing what we think we asked it to do. For example, if we wondered about the computation of M in the above example, we could have inserted 65 PRINT M to have the values of M printed.

SECTION 2 – ADVANCED BASIC

This section explains the following specialized aspects of the BASIC language:

- Logical Operators
- Special Functions
- Matrices

LOGICAL OPERATORS

In addition to the arithmetic operations and relationships, and mathematical functions already discussed (section 1.2.1), BASIC uses the **Boolean logical** *operators* AND, OR, and NOT. With the exception of NOT, which takes the following operand as its single argument, all of these are binary operators. In a formula, the binary operators have a lower priority than any of the arithmetic operators (↑, *, /, +, and −). Among the binary operators, the priority in ascending order is OR, AND, and the relational operators (all of equal priority). NOT has the same priority as unary + and unary −. Thus,

NOT A + B = C OR SGN(K) AND SGN(J)

is equivalent to

(((NOT A) + B) = C) OR (SGN(K) AND SGN(J))

The relational operators take algebraic numbers as arguments and return 0 (false) or 1 (true) according to the relationship existing between their arguments. Thus, 3 < 2 evaluates to 0 and A ≠ 0 evaluates to 1 if A has a nonzero value. The Boolean operators consider their arguments as zero (false) or nonzero (true) and return 0 and 1 as follows:

AND

Argument 1	Argument 2	Result
nonzero	nonzero	1
nonzero	zero	0
zero	nonzero	0
zero	zero	0

OR

Argument 1	Argument 2	Result
nonzero	nonzero	1
nonzero	zero	1
zero	nonzero	1
zero	zero	0

NOT

Argument	Result
nonzero	0
zero	1

Thus, 3 AND 1 evaluates to 1 while NOT -3 AND 0 evaluates to 0. It is important to realize the A < B < C is not equivalent to A < B AND B < C. If A = -3, B = -2, and C = 1/2, the former evaluates as (-3 < -2) < 1/2 or 0 (false), whereas the latter evaluates as (-3 < -2) AND (-2 < 1/2) or 1 (true).

SPECIAL FUNCTIONS

This section explains the special functions INT, RND, and SGN (listed in section 1.2.1), and the use of DEF to define other functions.

INT (Integer) Function

The INT function computes the value of x expressed by the algebraic notation as [x]. It gives the greatest integer not greater than x for $-32768 \leq x < 32768$. Thus, INT(2.35) = 2, INT(-2.35) = -3, and INT(12) = 12. Note that INT(X) = 32768 for $x \geqslant 32768$ and INT(X) = -32768 for $x \leqslant -32768$.

One use of the INT function is to truncate numbers. Use it to truncate to the nearest integer by writing INT(X + .5). This will truncate 2.9, for example, to 3 by finding INT(2.9 + .5) = INT(3.4) = 3. Thus, this function will truncate a number midway between two integers, up to the larger of the integers.

It can also be used to round to any specific number of decimal places. For example, INT(10 * X + .5)/10 rounds to one decimal place, and INT(10 ↑ D * X + .5)/10 ↑ D rounds to D decimal places.

RND (Randomize) Function

The RND function produces a normal distribution of random numbers between 0 and 1. The form of RND, RND(X) or RND(0), requires an argument, although the argument has no significance. The argument can be a constant or a previously defined variable. The example below produces 20 random six-digit decimals.

Program

```
READY
10    FOR L = 1 TO 20
20    PRINT RND(0);
30    NEXT L
40    END
      RUN
```

Result

```
RUN
6.80170E-02   .240643   .417191   .192204   .701485   .705105
1.80673E-02   .460499   .87426   .34657   .812546   .146031
.723061   .473629   .249848   .182393   .697106   1.98793E-02
.122941   .221928   READY

RUN
6.80170E-02   .240643   .417191   .192204   .701485   .705105
1.80673E-02   .460499   .87426   .34657   .812546   .146031
.723061   .473629   .249848   .182393   .697106   1.98793E-02
.122941   .221928   READY
```

Note that the second RUN gives exactly the same random numbers as the first. This greatly facilitates the debugging of programs that use the random-number generator.

To produce 20 random one-digit integers, change line 20 to

```
20    PRINT INT(10*RND(0)),
```

Result

```
0     2     4     1     7     7     0     4     8     3     8     1
7     4     2     1     6     0     1     2     READY
```

To vary the type of random numbers (for example, to obtain 20 random numbers ranging from 1 to 9 inclusive), change line 20 to

```
20    PRINT INT(9*RND(0) + 1);
```

Result

```
1     3     4     2     7     7     1     4     8     4     8     2
7     5     3     2     7     1     2     2     READY
```

To obtain random numbers that are integers between 5 and 24 inclusive, change line 20 to

```
20    PRINT INT(20*RND(0) + 5);
```

Result

```
6     9     13    8     19    19    5     14    22    11    21    7
19    14    9     8     18    5     7     9     READY
```

SGN (Sign) Function

The SGN function assigns the value 1 to any positive number, 0 to zero, and −1 to any negative number. Thus, SGN(7.23) = 1, SGN(0) = 0, and SGN(−.2337) = −1.

DEF (Define) Function

In addition to standard functions, any other function can be defined with DEF. The name of the defined function comprises three letters, the first two of which are FN. A total of 26 functions can be defined, e.g., FNA, FNB, etc.

For example, DEF can be used in a program where the function $\exp(-x^2 + 5)$ is needed frequently:

```
30    DEFFN E(X) = EXP (-X↑2+5)
```

Various values of the function can be called by writing FNE(.1), FNE(3.45), FNE(A+2), etc. Such a definition can be a great time-saver to produce values of some function for a number of different values of the variable.

DEF can occur anywhere in the program, and the expression to the right of the equal sign can be any legal expression. It can contain a combination of other functions, including those defined by other DEF statements. It can involve variables other than the one denoting the argument of the function.

For example, assume FNR is defined by

```
70    DEFFN R(X) = SQR (2+LOG (X)-EXP (Y*Z) * (X+SIN(2*Z)))
```

If values have been previously assigned to Y and Z, FNR(2.7) can be requested. New values can be assigned to Y and Z before the next use of FN.

The use of DEF is generally limited to those functions whose values can be computed within a single BASIC statement. More complicated functions or parts of a program are coded as subroutines accessible to GOSUB statements (section 3.8).

MATRICES

It is often convenient to interpret doubly subscripted arrays as **matrices**. Although matrix computations can be worked out using conventional BASIC statements, the language provides the following 12 matrix (MAT) statements to simplify program writing and

increase the power of the language.

MAT c = ZER	Fill matrix c with zeros
MAT c = CON	Fill matrix c with ones
MAT c = IDN	Define c as an identity matrix
MAT PRINT a, b; c	Print three matrices, in this example with a and c in the regular format and b closely packed (section 3.9.2)
MAT b = a	Set matrix b equal to matrix a
MAT c = a + b	Add the two matrices a and b and place the result in matrix c
MAT c = a - b	Subtract matrix b from matrix a and place the result in matrix c
MAT c = a * b	Multiply matrix a by matrix b and place the result in matrix c
MAT c = TRN(a)	Transpose matrix a and place the result in matrix c
MAT c = (k) * a	Multiply matrix a by the number or expression k (which must be in parentheses) and place the result in matrix c
MAT c = INV(a)	Invert matrix a and place the result in matrix c

BASIC matrices adhere to the following convention: if a MAT statement specifies a matrix having the dimensions m-by-n, the rows are numbered 1, 2, ..., M and the columns 1, 2, ..., N.

MAT statements are used in conjunction with dimension (DIM) statements that indicate the *maximum dimensions* of the matrices and cause the computer to save sufficient space. (The assumed dimensions of 10 rows by 10 columns for matrices without DIM statements do not apply for matrices involved in matrix computations. These must always be described by a DIM statement.) For example, to save space for any matrix up to and including 20 rows and 35 columns, we write

```
DIM M(20,35)
```

The *actual dimensions* of a matrix can be defined either when first established (by using a DIM statement), or by one of the four matrix statements MAT READ, MAT ZER, MAT CON, or MAT IDN. Thus, to read a 20-by-7 matrix for x, write

```
10    DIM X(20,7)
      .
      .
      .
50    MAT READ X
```

To read a 17-by-30 matrix for y within maximum dimensions of 20-by-35, write

```
10    DIM Y(20,35)
      .
      .
      .
50    MAT READ Y(17,30)
```

The elements of a matrix are stored by column in ascending locations in memory, using two computer words for each element. Thus, the matrix dimensioned as DIM A(3,3) is structured and stored as follows:

	Columns		
Rows	A(1,1)	A(1,2)	A(1,3)
	A(2,1)	A(2,2)	A(2,3)
	A(3,1)	A(3,2)	A(3,3)

The elements would be stored in the following order:

Element Position	**Memory Location**	**Element**
1	m	A(1,1)
2	m + 2	A(2,1)
3	m + 4	A(3,1)
4	m + 6	A(1,2)
5	m + 8	A(2,2)
6	m + 10	A(3,2)
7	m + 12	A(1,3)
8	m + 14	A(2,3)
9	m + 16	A(3,3)

Given the statement DIM A(M,N), the location m of any element A(i,j) of the matrix, with respect to the first element A(1,1), is given by:

$$m = [\text{location of } A(1,1)] + 2[M(i-1) + (j-1)]$$

The three statements:

```
MAT M = ZER
MAT M = CON
MAT M = IDN
```

set up the matrix M filled with zeros, filled with ones, or as an identity matrix, respectively. Each acts as MAT READ as far as the dimensioning of the matrix is concerned. For example,

```
MAT M = CON(7,3)
```

sets up a 7-by-3 matrix full of ones, but

```
MAT M = CON
```

sets up a matrix, also full of ones, according to dimensions previously specified by a DIM statement. Thus,

```
10    DIM M(20,7)
20    MAT READ M(7,3)
      .
      .
      .
35    MAT M = CON
      .
      .
      .
70    MAT M = ZER(15,7)
```

will first read in a 7-by-3 matrix for M and then set up a 7-by-3 matrix of ones for M as specified in line 20. This results in an error message because line 70 calls for 105 components in a matrix limited to 21 components by line 20. The original dimensions can be exceeded, however, provided the total number of components is within the set limit, e.g.,

```
90    MAT M = ZER(25,5)
```

The MAT PRINT statement (see also section 3) prints matrix components row by row across the page. Spacing between elements is controlled by typing commas or semicolons after each component, where commas space the printing and semicolons close-pack it. Each row starts on a new line. Rows containing more components than can be printed on one line are continued on the next line. Thus, the statement

```
MAT PRINT A, B; C
```

prints the matrices A and C in the normal format of five components per line, and matrix B closely packed with up to 12 components per line.

Vectors

A singly subscripted array can be interpreted as a column vector. Vectors can be used in place of matrices as long as the above rules are followed. Since a vector like V(J) is treated as a column vector by BASIC, a row vector must be specified as a matrix with one row, e.g.,

```
DIM X(7), Y(1,5)
```

introduces a seven-component column vector and a five-component row vector. A column vector is printed one element per line with double spacing between lines. A row vector is printed as specified by the statement, e.g., where V is a row vector,

```
MAT PRINT V,
```

prints V as a row vector, five components to the line, while

```
MAT PRINT V;
```

prints V as a row vector, twelve components to the line.

Manipulating Matrices

To set up a matrix B identical to the matrix A, provided that the dimensions previously assigned to B are the same as those of A, write

```
MAT B = A
```

If matrices A, B, and C have the same dimensions, operations such as

```
MAT C = A + B
MAT C = A - B
```

are legal. The indicated operation is performed and stored in C. Only one operation per statement is allowed, e.g., to perform MAT D = A + B - C two statements are required.

In the multiplication operation

```
MAT C = A * B
```

the number of columns in A is equal to the number of rows in B, the number of rows in C is equal to the number of rows in A, and the number of columns in C is equal to the number of columns in B. For example, if A is l-by-m, and B is m-by-n, then C is l-by-n.

(Note that even when MAT A = A + B is legal, MAT A = A * B results in nonsense because, in multiplying matrices, components required to complete the computation are destroyed before they can be used, whereas, in addition, the results are stored immediately. However, MAT B = A * A is legal provided A is a square matrix.)

Matrices can also be multiplied by constants. In the operation

```
MAT C = (k) * A
```

each component of the matrix A is multiplied by k to form the components of the matrix C. The constant k, which is enclosed in parentheses, can be a number or an expression. The statement MAT A = (k) * A is legal.

To transpose the matrix A, write

```
MAT C = TRN(A)
```

where matrix C is matrix A transposed. Thus, if A is m-by-n, C is n-by-m. Since a matrix is destroyed by transposition, it cannot be transposed into itself, i.e., dimensions cannot be reversed by writing MAT A = TRN(A).

To invert the square matrix A, write

```
MAT C = INV(A)
```

where matrix C is the inversion of the square matrix A. Like transposition, a matrix cannot be inverted into itself.

Sample Matrix Programs

EXAMPLE 1: This program reads in A and B in line 30 and in so doing sets up the correct dimensions. Dimensions for C are set up in line 35. Then, in line 40, A + A is computed and the answer is called C. Note that the data in line 90 result in A being 2-by-3 and B being 3-by-3. Both MAT PRINT formats are illustrated and one method of labeling a matrix print is shown.

Program

```
READY
10    DIM A(15,15),B(15,15),C(15,15)
20    READ M, N
30    MAT READ A(M,N),B(N,N)
35    MAT C = ZER(M,N)
40    MAT C = A + A
50    MAT PRINT C;
60    MAT C = A*B
70    PRINT
```

```
75    PRINT "A*B="
80    MAT PRINT C,
90    DATA 2, 3
91    DATA 1, 2, 3
92    DATA 4, 5, 6
93    DATA 1, 0, -1
94    DATA 0, -1, -1
95    DATA -1, 0, 0
99    END
```

Result

```
      RUN
       2     4     6
       8     10    12
A*B =
      -2    -2    -3
      -2    -5    -9
```

EXAMPLE 2: This program inverts an n-by-n Hilbert matrix:

$$\begin{matrix} 1 & 1/2 & 1/3 & \cdots & 1/n \\ 1/2 & 1/3 & 1/4 & \cdots & 1/n+1 \\ 1/3 & 1/4 & 1/5 & \cdots & 1/n+2 \\ \cdot & \cdot & \cdot & \cdots & \\ \cdot & \cdot & \cdot & \cdots & \\ \cdot & \cdot & \cdot & \cdots & \\ 1/n & 1/n+1 & 1/n+2 & & 1/n-1 \end{matrix}$$

Ordinary BASIC instructions are used to set up the matrix in lines 50 to 90. Note that this occurs after correct dimensions have been declared. Then a single instruction results in the computation of the inverse matrix, and one more instruction prints it. In this example, we have supplied 4 for n in the DATA statement and have made a run for this case.

Program

```
READY
5     REM THIS PROGRAM INVERTS AN N-BY-
                                        N HILBERT MATRIX
10    DIM A(20,20), B(20,20)
20    READ N
30    MAT A = CON(N,N)
40    MAT B = CON(N,N)
50    FOR I = 1 TO N
60    FOR J = 1 TO N
70    LET A(I,J) = 1/(I+J-1)
```

```
80    NEXT J
90    NEXT I
100   MAT B = INV(A)
110   PRINT
115   PRINT "INV(A) ="
120   PRINT
125   MAT PRINT B;
190   DATA 4
199   END
RUN
```

Result

```
INV(A) =
15.9885     -119.877    239.713     -139.816
-119.877    1198.69     -1696.94    1678.04
239.713     -2696.94    6472.84     -4195.42
-139.816    1678.04     -4195.42    2797.07
```

Note: Because of severe rounding errors, Hilbert matrices are not inverted beyond n = 7.

SECTION 3 – STATEMENTS IN BASIC

This section explains each type of BASIC statement and illustrates its use:

- READ statement
- DATA statement
- DIM (dimension) statement
- MAT (matrix) statements
- LET statement
- FOR statement
- NEXT statement
- IF/THEN statement (conditional GO TO statement)
- GO TO statements
  - Unconditional GO TO statement
  - Computed GO TO statement
- GOSUB (go to subroutine) statements
  - Unconditional GOSUB statement
  - Computed GOSUB statement
  - GOSUB statement with parameters
- RETURN statement
- SUB (subroutine) statement
- PRINT statement
- INPUT statement
- RESTORE statement
- REM (remark) statement

- CALL statement
- WAIT statement
- STOP statement
- END statement

READ and DATA Statements

The READ statement has the format

number **READ var**,*var*,...

and the DATA statement has the format

number **DATA num**,*num*,...

where

var	is a variable
num	is a number

The sequences of variables and numbers in these statements can contain any number of items as long as the statement itself does not exceed 72 characters.

A READ statement assigns values obtained from a DATA statement to the listed variables. Neither statement is used without one of the other type. A READ statement causes the variables listed in it to be given, in order, the next available numbers in the collection of DATA statements. Before the program is run, the computer takes all of the DATA statements in the order in which they are numbered and creates a large data block. Each time a READ statement is encountered anywhere in the program, the data block supplies the next available number or numbers. If the data block runs out of data with a READ statement still asking for more, the program is assumed to be done and we get an OUT OF DATA error message:

```
ERROR 56 IN LINE nn
```

Since we have to read in data before working with it, READ statements normally occur near the beginning of a program. The location of DATA statements is arbitrary, as long as they are numbered in the correct order. A common practice is to collect all DATA statements and place them just before the END statement.

Examples:

```
150  READ X, Y, Z, X1, Y2, Q9
330  DATA 4, 2, 1.7
340  DATA 6.734E-3, -174.321, 3.14159265
```

```
234   READ B(K)
263   DATA 2, 3, 5, 7, 9, 11, 10, 8, 6, 4
10    READ R(I,J)
440   DATA -3, 5, -9, 2.37, 2.9876, -437.234E-5
450   DATA 2.765, 5.5576, 2.3789E2
```

Remember that only numbers are put in a DATA statement, and that 15/7 and $\sqrt{3}$ are expressions, not numbers.

DIM (Dimension) Statement

The DIM statement has the format

number **DIM array**

where

number is the line number of the statement
array is the name of the array being dimensioned followed by its subscript(s) in parentheses

The DIM statement is required for defining an array having any subscript greater than 10. The maximum subscript allowed is 255.

Examples:

```
20    DIM H (35)
35    DIM Q (5,25)
```

MAT (Matrix) Statement

The MAT statement is explained in the section on matrices.

LET Statement

This statement has the format

number **LET var = exp**

or the format

number **LET var =** *voa* **= ... = exp**

where

number is the line number of the statement
var is a variable
exp is a number or an expression
voa is a variable or an array

This statement assigns the value of the number or expression to one or more variables or arrays.

Examples:

```
100   LET X = X + 1
200   LET W7 = (W-X4↑3)*(Z-A/(A-B))-17
333   LET X = Y3 = A(3,1) = 1
900   LET W = Z = 3*X-4*X↑2
```

FOR and NEXT Statements

The FOR statement has the format

number **FOR var = expi TO expf** *STEP exps*

and the NEXT statement has the format

number **NEXT var**

where

number	is the line number of the statement
var	is a simple (nonsubscripted) variable, identical in both the FOR and NEXT statements of the couplet
expi	is a number or expression whose value is the initial value of **var**
expf	is a number or expression whose value is the final value of **var**
exps	is a number or expression whose value is the increment between successive values of **var** between **expi** and **expf** (if omitted from the statement, *exps* is assumed to be +1)

The FOR statement enters a loop and the NEXT statement exits from the loop, directing the computer back to the FOR statement until **expf** is reached. At this point the NEXT statement allows the computer to exit from the loop and pass to the next statement in the program.

Specifications and restrictions in the use of loops, as well as examples of FOR and NEXT statements, are given in the discussion of loops.

IF/THEN Statement

This statement has the format

number **IF exp** *rel exp* **THEN next**

where

number	is the line number of the IF/THEN statement
exp	is a mathematical expression or number
rel	is an arithmetic operation or relationship

exp is a mathematical expression or number used if and only if *rel* is present

next is the line number of the statement to be executed next if the preceding conditions are met

The conditions of an IF/THEN statement are met when the logical argument **exp** *rel exp* is true or if the single mathematical expression **exp** is nonzero. Any expression can be considered a logical argument by evaluating the numbers represented in it by the logical constants false = 0 and true = not 0.

Examples:

```
40   IF SIN (X) < = M THEN 80
```

where the computer jumps to line 80 if the sine of x is less than or equal to m, but otherwise goes to the next line after 40.

```
20   IF G = 0 THEN 65
```

where the computer jumps to line 65 if G is zero, but otherwise goes to the next line after 20.

```
35   IF A THEN 83
```

where the computer jumps to line 83 if A is nonzero, but otherwise goes to the next line after 35.

```
85   IF A + B - 5 THEN 302
```

where the computer jumps to line 302 if the value of the expression is nonzero, but otherwise goes to the next line after 85.

```
90   IF -2 THEN 200
```

where the computer always jumps to line 200 since the value of the expression is always nonzero, i.e., such a statement is equivalent to an unconditional GO TO statement.

GO TO Statements

There are two types of GO TO statements. The **unconditional GO TO statement** causes the program execution to jump to the specified line every time the statement is encountered. This statement has the format

number GO TO next

where

number	is the line number of the GO TO statement
next	is the line number of the statement to be executed next

The **computed GO TO statement** specifies several lines for the jump. The one selected depends on the value n of an expression in the statement, where the line number chosen is the nth line number in the statement. This statement has the format

number **GO TO exp OF next,next,***next,...*

where

number	is the line number of the GO TO statement
exp	is an expression
next	is a line number of one of the statements that, depending on the value of **exp**, is to be executed next

Thus, **exp** is evaluated and the answer truncated to the integer value n that determines the **next** to be used, e.g., if n = 2, the second **next** is the valid line number.

The IF/THEN statement (section 3.6) is also called the **conditional GO TO statement** because the program jumps to the specified line only if a certain relationship exists.

Examples: *Unconditional GO TO Statement:*

```
150  GO TO 75
```

where the next statement to be executed in the one in line 75.

Computed GO TO Statement:

```
160  GO TO I-3 OF 10,30,100
```

where the next statement to be executed is the one in line 30 when I = 5. since I-3 = 2, selecting the second line number in the series.

Conditional GO TO Statement: See IF/THEN statement (section 3.6).

GOSUB, RETURN, and SUB Statements

There are three types of SOSUB (Go To Subroutine) statements. All of them direct the computer to a subroutine according to the specifications of the statement, and all of them are used with the **RETURN statement**, which has the format

number **RETURN**

where **number** is the line number of the RETURN statement. The RETURN statement is

the exit from the subroutine and returns the computer to the first line number greater than that of the calling GOSUB statement. RETURN is the only exit from a subroutine, i.e., GO TO or IF/THEN statements cannot be used to exit from a subroutine. There can be more than one RETURN statement in a subroutine, but only one of them can be used on any given pass through the subroutine.

Subroutines can be nested, i.e., GOSUB statements can be used inside subroutines to call sub-subroutines.

The **unconditional GOSUB statement** causes the program execution to jump to the specified line every time the statement is encountered. This statement has the format

number **GOSUB subr**

where

number is the line number of the GOSUB statement
subr is the line number of the first statement in the subroutine to be executed next

The **computed GOSUB statement** specifies several lines for the subroutine jump. The one selected depends on the value n of an expression in the statement, where the line number chosen is the nth line number in the statement. This statement has the format

number **GOSUB exp OF subr,subr,***subr,...*

where

number is the line number of the GOSUB statement
exp is an expression
subr is the line number of one of the statements that, depending on the value of **exp**, is the first statement of the subroutine to be executed next

Thus, **exp** is evaluated and the answer truncated to the integer value n that determines the **subr** that will be used, e.g., if n = 2, the second **subr** is the valid line number of the first statement in the subroutine selected.

The **GOSUB statement with parameters** enters parameter values in the subroutine specified. The statement has the format

number **GOSUB subr,param,***param,...*

where

number is the line number of the GOSUB statement
subr is the line number of the first statement in the subroutine to be executed next, i.e., that of the SUB statement
param is a parameter value to be entered into the subroutine by the SUB statement

The GOSUB statement with parameters is always used with a **SUB statement** that has the format

subr **SUB var,***var,...*

where

subr is the line number of the SUB statement and is identical with the **subr** in the corresponding GOSUB statement

var is a variable

The parameter values in the GOSUB statement are assigned to the corresponding variables in the SUB statement; i.e., the first parameter value is assigned to the first variable, etc. The effect of a SUB statement or the definition of variables is removed upon execution of the corresponding RETURN statement. A subroutine defined by a SUB statement can be entered retrogressively (see example below).

Examples: Unconditional GOSUB Statement: This program for finding the greatest common denominator (GCD) of three integers using the Euclidean algorithm illustrates the use of the unconditional GOSUB statement. The first two numbers are selected in lines 30 and 40 and their GCD is determined in the subroutine, lines 200 through 310. This GCD is called X in line 60, the third number is called Y in line 70, and the subroutine is entered from line 80 to find the GCD of these two numbers. The resulting GCD is the greatest common divisor of the three given numbers and is printed with them in line 90.

Program

```
READY
10    PRINT "A", "B", "C", "GCD"
20    READ A, B, C
30    LET X = A
40    LET Y = B
50    GOSUB 200
60    LET X = G
70    LET Y = C
80    GOSUB 200
90    PRINT A, B, C, G
100   GO TO 20
110   DATA 60, 90, 120
120   DATA 38456, 64872, 98765
130   DATA 32, 384, 72
200   LET Q = INT(X/Y)
210   LET R = X - Q*Y
220   IF R = 0 THEN 300
230   LET X = Y
240   LET Y = R
250   GO TO 200
300   LET G = Y
310   RETURN
320   END
RUN
```

Result

```
A             B             C             GCD
60            90            120           30
38456         64872         98764         4
32            384           72            8
ERROR 56 IN LINE 20
```

Computed GOSUB Statement: The expression in the statement is evaluated and truncated to the integer n, and program execution jumps to the nth line number in the line number list. If, when z = 1.68, we want the program execution to jump to line 55, we can write

```
20    GOSUB Z+1 OF 30,55,70
```

where the expression is computed to be equal to 2.68 and is truncated to the integer 2, thus selecting the second line number. In this case, the computed GOSUB results in a jump like that obtained with the unconditional GOSUB

```
20    GOSUB 55
```

GOSUB Statement with Parameters: To pass the two parameter values 5 and 10 into a subroutine defined by a SUB statement in line 100, we can write

```
20    GOSUB 100, 5, 10
```

or, the same transfer results when we use a variable j when its value is 6 and we write

```
20    GOSUB 100, 5, J+4
```

GOSUB Statement with Parameters and SUB Statement for Retrogressive Subroutine Entry: This program shows the use of these statements for a simple incrementation.

Program

```
LIST
10    REM SHOW USE OF GOSUB WITH PARAMETER TRANSFER
20    LET N = 5
30    GOSUB 100, N
40    PRINT "MAIN PROGRAM"; N
50    END
100   SUB N
105   REM THIS EXAMPLE USES A RETROGRESSIVE SUBR. ENTRY
110   PRINT "SUBROUTINE GOT"; N
120   IF N = 0 THEN 140
130   GOSUB 100, N-1
140   RETURN
9999  END
RUN
```

Result

```
SUBROUTINE GOT 5
SUBROUTINE GOT 4
SUBROUTINE GOT 3
SUBROUTINE GOT 2
SUBROUTINE GOT 1
SUBROUTINE GOT 0
MAIN PROGRAM 5
READY
```

PRINT Statements

The PRINT statements control the output and format of the results of BASIC programs.

General Types

There are four common uses of PRINT statement:

- To print the results of computations
- To print comments
- To print a combination of the above
- To skip a line

Each of these uses requires a particular format of PRINT statement.

To print the results of computations, use the format

number **PRINT exp**,*exp*,...

where

number is the line number of the statement
exp is a variable or expression whose computed value is to be printed (up to 5 values per line)

The variables used must already have been given values.

Examples: To print the value of x and the value of its square root, we can write

```
100  PRINT X, SQR(X)
```

To print the values of the five expressions x, y, z, $b^2 - 4ac$, and e to the power a − b, we can write

```
135  PRINT X, Y, Z, B*B-4*A*C, EXP(A-B)
```

To print comments, use the format

number PRINT "comment"

where

number is the line number of the statement
comment is the material to be printed

Within **comment** blanks will be observed by the computer. If several comments are to be printed on one line, e.g., for column headings, use the format

number PRINT "comment", "comment",

where each **comment** is enclosed within quotation marks. In no case are the quotation marks printed When used for column headings, the printout automatically aligns the headings and the columns because the commas specify that the next heading be printed in the next zone (section 3.9.2).

Examples:

```
100  PRINT "NO UNIQUE SOLUTION"
430  PRINT "X VALUE", "SIN", "RESOLUTION"
```

To print a combination of results and comments, use the PRINT statement with the comments in quotation marks, and with the variables or expressions whose values are to be printed given as above.

Examples: Where x = 625, we can write

```
15   PRINT "THE VALUE OF X IS" X
30   PRINT "THE SQUARE ROOT OF" X; "IS" SQR(X)
```

and obtain the printout

```
THE VALUE OF X IS 625
THE SQUARE ROOT OF 625 IS 25
```

Note that no terminator (semicolon or comma) is required after quotation marks, but they are required after variables or expressions (except as the final item in the statement). The comma causes the item that follows to be printed in the next zone, while a semicolon causes it to be printed closed-up (section 3.9.2).

To skip a line, use the format

number PRINT

where **number** is the line number of the statement.

This statement simply requests that the computer print nothing and then activate the carriage return, i.e., skip a line.

Manipulating the Printing Format

The Teletype line is divided into five **printing zones**, starting at positions 0, 15, 30, 45, and 60, respectively. A **terminator** (comma or semicolon) controls the use of these zones. A *comma (,)* moves printing to the next printing zone, or, if the fifth printing zone has been filled, to the first printing zone of the next line. A *semicolon (;)* produces more compact output since it inhibits spacing between printing zones, acting only to separate quantities to be printed (e.g., A+B;C/D), or to suppress a carriage return at the end of a print statement.

Spacing within a printing zone depends on the value and type of the number being printed. A number is always printed in a zone larger than it needs, and is left-justified in that zone. The zone size is determined as follows:

Value of Number	Type of Number	Format of Zone
$-999 \leq n \leq +999$	Integer	VV xxx V
$-999999 \leq n \leq -1000$ $+1000 \leq n \leq +999999$	Integer	Λxxxxxx ΛΛΛ -99999 ⩽ n ⩽-1000 INT +1000 ⩽ n ⩽ 99999
$0.1 \leq n \leq 999999.5$	Real (normal range)	VVVV xxxxxxx V -999999 ⩽ n ⩽-10000 INT +10000 ⩽ n ⩽-999999 (decimal point printed as one of x's; trailing zeros suppressed)
$n < 0.1$ $999999.5 < n$	Large integer or real (extreme range)	≙ x.xxxxx E ± ee ΛΛΛ

The carat (Λ) represents a space typed on the Teletype.

For example, for the program

```
10    FOR I = 1 TO 15
20    PRINT I
30    NEXT I
40    END
RUN
```

the Teletype prints 1 at the beginning of a line, 2 at the beginning of the next line, etc., up to 15 on the 15th line.

By changing line 20 to read

```
20    PRINT I,
RUN
```

the numbers are printed in zones, reading

```
1          2          3          4          5
6          7          8          9          10
11         12         13         14         15
```

To print the numbers in more tightly packed zones, replace the comma in line 20 with a semicolon

```
20    PRINT I;
RUN
```

and the result is

```
1    2    3    4    5    6    7    8    9    10
11   12   13   14   15
```

A character string in quotation marks is printed just as it appears. The end of a PRINT line always signals a new line unless a comma or a semicolon is the last symbol. Thus, the statement

```
50    PRINT X, Y
```

prints the two numbers and returns to the next line, while the statement

```
50    PRINT X,Y,
```

prints these two values but does not return. The next number is printed in the third zone, following the values of X and Y in the first two zones.

Since the end of a PRINT statement signals a new line,

```
250   PRINT
```

skips one line. This can be used to put a blank line in the program to allow vertical spacing of the results. It can also be used to complete a partially filled line:

```
50    FOR M = 1 TO N
110   FOR J = 1 TO M+1
```

```
120   PRINT B(M,J);
130   NEXT J
140   PRINT
150   NEXT M
```

This program prints B(1,1) followed by B(1,2). Without line 140, the Teletype would continue to print B(2,1), B(2,2), and B(2,3) on the same line, and then B(3,1), B(3,2), etc. Line 140 directs the Teletype to start a new line after printing the B(1,2) value corresponding to M = 1, and again after printing the value of B(2,3) corresponding to M = 2, etc.

The instructions

```
50    PRINT "VARIAN BASIC ";
51    PRINT "LANGUAGE COMPILER"
```

print

```
VARIAN BASIC LANGUAGE COMPILER
```

Output formatting can be controlled even further by use of the function TAB. Insertion of TAB(17) causes the Teletype to move to column 17 as if a tab had been set there. For this purpose, line positions are numbered 0 through 71.

TAB can contain any expression as its argument. The value of the expression is computed, truncated, and its integer part taken. The Teletype then moves forward to this position. If the position has already been passed, the TAB is ignored. If the result is greater than 71, the Teletype moves to position 0 of the next line.

For example, to insert the following line in a loop, we write

```
PRINT X; TAB(12); Y; TAB(27); Z
```

This prints the X value in column 0, the Y value in column 12, and the Z value in column 27.

A comma following a TAB clause has no effect on Teletype positioning. The statement

```
PRINT TAB(7), A+B
```

prints the value of A + B starting at position 7. The statement

```
PRINT Z, A+B
```

prints the value of A + B at position 15 (second printing zone).

The following rules for the printing of numbers will aid in interpreting printed results:

a. If the number is an integer with a value from −999999 to +999999, inclusive, the decimal point is not printed.

b. If the number is real and has an absolute value between 0.1 and 999999.5, it is rounded to six digits and printed with a decimal point. Trailing zeros after the decimal point are suppressed.

c. A number either greater than 999999.5 or less than 0.1 is rounded to six places. The Teletype then prints a space (if positive) or a minus sign (if negative), the first digit, the decimal point, the next five digits, the letter E (exponent), the sign of the exponent, and the value of the exponent. For example, 3.243,756 is printed as 3.243756E+6.

The following program to print the powers-of-two shows how numbers are printed.

Program

```
READY
10    FOR N = -5 TO 30
20    PRINT 2↑N;
30    NEXT N
40    END
RUN
```

Result

```
3.12501E-02    6.25002E-02    .125    .25    .5    1
1.99999    3.99999    7.99997    15.9999    31.9998    63.9995
127.999    255.998    511.994    1023.99    2047.98    4095.94
8191.88    16383.7    32767.4    65535    131069    262138
524277    1.04855E+06    2.09790E+06    4.19422E+06    8.38839E+06
1.67768E+07    3.35536E+07    6.71068E+07    1.34213E+08    2.68427E+08
5.36854E+08    1.07370E+09    READY
```

INPUT Statement

The INPUT statement has the format

number **INPUT var**,*var*,...

where

number is the line number of the statement
var is a variable

The INPUT statement acts as a READ statement but does not draw data from DATA

statements. Instead, it asks the user to supply the required data by outputting a question mark. The user types the data, separating each number from the next with a comma, and presses the RETURN key on the Teletype.

INPUT should be used only when small amounts of data are to be entered, or when it is necessary to enter data during the running of the program, since data entry via INPUT is slow. Furthermore, *data entered via INPUT statements are not saved with the program.* Numbers used with INPUT must not exceed nine digits.

To retake control from the INPUT processor, type a plus (+).

Examples: If the user is to supply values for x and y, type

```
40    INPUT X, Y
```

before the first statement that uses either variable. When the computer encounters this statement, it types a question mark. Type two numbers, separated by a comma and press the RETURN key. The computer then goes on with the rest of the program.

Frequently, an INPUT statement is combined with a PRINT statement to ensure that the user knows what the question mark is asking for. You might type, for example,

```
20    PRINT "YOUR VALUES OF X, Y, AND Z ARE";
30    INPUT X, Y, Z
```

and the computer will type

```
YOUR VALUES OF X, Y, AND Z ARE?
```

(Without the semicolon at the end of line 20, the question mark would have been printed on the next line.)

RESTORE Statement

The RESTORE statement has the format

number **RESTORE**

where **number** is the line number of the statement.

The RESTORE statement permits the reuse of data within a program. When RESTORE is encountered in a program, the computer restores the data-block pointer to the first data number. A subsequent READ statement then starts the reading of the data again from the first data item.

If the desired data items are preceded by unwanted data, use extra READ statements to pass over these numbers.

Example: This program reads the data, restores the data-block pointer, and rereads the data. Note the use of line 570 to pass over the already-known value of n.

```
100   READ N
110   FOR I = 1 TO N
120   READ X
        .
        .
        .
200   NEXT I
        .
        .
        .
        .
560   RESTORE
570   READ X
580   FOR I = 1 TO N
590   READ X
```

REM (Remark) Statement

The REM statement has the format

number REM comment

where

number is the line number of the statement
comment is any comment desired in the listing

The comment in the statement is printed in the listing just as written, including blanks. It is, however, otherwise ignored by the processor.

The line number of a REM statement can be used in a GO TO or IF/THEN statement.

Examples:

```
100   REM INSERT DATA IN LINE 900-998.  THE FIRST
110   REM NUMBER IS N, THE NUMBER OF POINTS.  THEN
120   REM THE DATA POINTS THEMSELVES ARE ENTERED, BY

200   REM THIS IS A SUBROUTINE FOR SOLVING EQUATIONS
        .
        .
        .
300   RETURN
        .
        .
```

```
520  GOSUB 200
```

CALL Statement

The CALL statement has the format

number **CALL asubr**,*parameter*,*parameter*,...

where

number is the line number of the statement
asubr is the name of an assembly-language subroutine (1-6 alphanumeric characters)
parameter is a variable, number, or expression

The CALL statement links absolute assembly-language subroutines to BASIC. Execution of the CALL statement passes control to an assembly-language program appended to the BASIC system. For internal design considerations of CALLed subroutines, see section 6.

Results can be returned to the BASIC program through the *parameters. Care is necessary to prevent an constant from being placed in a position in the CALL statement where the subroutine attempts to return a value. This results in the value of the constant being changed throughout the BASIC interpreter.*

Examples:

```
100  CALL SUBA, A/D, X, 3
200  CALL RESET
883  CALL POLY, A+3, B+4, C*D↑5, E
```

WAIT Statement

The WAIT statement has the format

number **WAIT delay**

where

number is the line number of the statement
delay is a number, variable, or expression whose value gives the number of milliseconds delay introduced into the program at this point (maximum value 32,767)

STOP Statement

The STOP statement has the format

number **STOP**

where **number** is the line number of the STOP statement.

The STOP statement stops the computer execution of the program. Execution resumes with the next BASIC statement when the RUN or START key on the computer is pressed.

END Statement

The END statement has the format

number END

where **number** is the line number of the END statement.

The END statement returns control to the operator. A program can have more than one END statement, but it is possible for a program to terminate without reaching an END statement, e.g., when there is no more data to process.

SECTION 4 – USING THE BASIC SYSTEM

This section provides:

- The operating instructions for the BASIC system
- The control commands that the user inputs from the Teletype keyboard
- An explanation of the two BASIC operating modes: program mode and calculator mode

OPERATING INSTRUCTIONS

The minimum hardware configuration for using BASIC is a Varian 73 or 620-series computer with 8K of memory and a model 33/35 ASR Teletype.

Memory requirements for the various configurations are:

Basic BASIC	6224 words
BASIC with trig functions	6585 words
BASIC with matrix functions	7421 words
Binary Load/Dump Program (BLD II)	256 words
Debugging Program (AID II)	1024 words

In an 8K system, AID II cannot be loaded in addition to a complete BASIC program.

To operate the BASIC system:

a. Manually enter the bootstrap loader program (refer to the applicable system reference handbook).

b. Load the BASIC system tape with the Binary Load/Dump program provided.

c. Start the program at location 02. The program types:

```
PAPER TAPE:  TYPE 1 FOR HI SPEED, ELSE TTY
```
(continued)

d. If the high-speed paper tape reader/punch is desired, enter a 1. Any other entry assigns paper tape input/output to the Teletype. The program types:

```
MAT, TRIG: 1 TO SAVE BOTH, 2 TO SAVE TRIG, ELSE
DELETE BOTH
```

e. The lower boundary of the BASIC table space is selected here. The program space used by the matrix or trig function can be deleted and used by BASIC for working storage. If there is more than 8K or memory, the program types:

```
AID: 1 TO SAVE, ELSE WIPEOUT
```

The core space occupied by the utility program AID will be used by BASIC unless there is a request that it be saved. In an 8K system, AID is always lost. The program types:

```
READY
```

and waits for input from the Teletype.

f. Start the input, which can be either a BASIC statement or a control command.

If, in the process of typing a statement, you make a typing error and notice it immediately, you can correct it by pressing the backward arrow (shift key above the letter O). This deletes the character in the preceding space, and you can then type in the correct character. Pressing this key a number of times will erase from a line the characters in that number of preceding spaces. To delete all of the present line, press RUBOUT. Programs or data can be annotated by typing the remark and then deleting the line (as far as the system is concerned) with a RUBOUT. BASIC types a backslash to show that a line has been deleted.

After typing your complete program, type RUN, press the RETURN key. If the program is free of errors, the computer will run it and type out any results that you requested in your PRINT statements. This does not mean that your program is entirely correct, but that it has no errors of the type known as grammatical errors.

If there are grammatical errors, the computer types an error code as soon as each error is detected while the program is being typed. Errors detected after RUN are structural (loop-nesting, matching GOSUB and return) or arithmetic errors. A list of error codes with an interpretation of each is in appendix A.

If you receive an error message, correct the error by typing a new line with the correct statement. For example, to eliminate the statement on line 110 from a program, type 110 and then press the carriage return. To insert a statement between those on lines 60 and 70, give it a line number between 60 and 70.

If it is obvious while the computer is running that the answers are wrong, press any Teletype key and computation ceases. The system types:

```
READY
```

and you can make your corrections.

Example:

Program

```
READY
10    FOR N = 1 TO 7
20    PRINT N, SQR(N)
30    NEXT N
40    PRINT "DONE"
50    END
RUN
```

Result

```
1     1
2     1.41421
3     1.73205
4     2
5     2.23607
6     2.44948
7     2.64575
DONE
```

At all times, there is only one device from which the BASIC interpreter will accept information and only one device on which it will output data. These devices are fixed at program initialization time.

CONTROL COMMANDS

The following are BASIC control commands:

Any key	Stops program execution when the program is running. Returns control to the Teletype.
RUN nnn	Begins execution of the program starting at line nnn. If no digits (nnn) are entered, 1 is assumed.
LIST nnn	Outputs an up-to-date listing of the program on the currently active listing device starting at line nnn. Listing can be terminated by pressing any key.
PLIST nnn	Same as **LIST**, but the output device is the high-speed paper tape punch or Teletype, whichever was

PTAPE	Moves control to the high-speed paper tape reader or Teletype, whichever was selected.
SCRATCH	Deletes the current BASIC program from memory.

PROGRAM AND CALCULATOR MODES

The description of the BASIC system to this point has assumed that the definition and execution of programs has occurred at different times, i.e., that one entered a set of numbered BASIC statements that were checked for validity and stored, then entered the control command RUN to start execution of the program defined by this set of statements. This mode of operation is called the **program mode**.

Varian BASIC can also run in the **calculator mode**, which causes the computer to respond immediately to a BASIC statement, just like a desk calculator. In this mode, one does not enter a statement number, but only a statement, and the BASIC system executes it immediately. *Example:*

```
PRINT 5*4-3
17
```

SECTION 5 – ERROR MESSAGES

An error message is printed as soon as the error condition is detected. The format is as follows:

ERROR xx IN LINE nn

Error Code xx	Meaning
0	Hardware M/D option missing
1	Statement ends unexpectedly
2	Input exceeds 72 characters
3	System command not recognized (can be missing statement number)
4	Missing or incorrect statement type
5	Exponent of number is missing power
6	Symbol following MAT not recognized
7	LET statement has no store
9	Missing or incorrect function identifier in DEF
10	Missing parameter in DEF statement
11	Missing assignment operator
12	Missing THEN
13	Missing or incorrect FOR variable
14	Missing TO
15	Incorrect STEP in FOR statement
16	Called routine does not exist
17	Wrong number of parameters in CALL statement
18	Missing or incorrect constant in DATA statement
19	Missing or incorrect variable in READ statement
20	No closing quotation mark for PRINT string
21	Missing print delimiter or bad PRINT quantity
22	Illegal word follows MAT
23	Missing delimiter
24	Improper matrix function
25	No subscript where expected
26	Cannot invert or transpose matrix into itself
27	Missing multiplication operator
28	Improper matrix operator
29	Matrix cannot be both operand and result of matrix multiplication
30	Missing left parenthesis

Error Code xx	Meaning
31	Missing right parenthesis
32	Operand not recognized
33	Defined array missing subscript part
34	Missing array identifier
35	Missing or bad integer
36	Nonblank characters following statement's logical end
37	Out of storage during syntax phase
38	Paper tape reader not ready or EOF on paper tape
39	Doubly defined function
40	FOR statement has no matching NEXT statement
41	NEXT statement has no matching FOR statement
42	Formal parameter finds no actual parameter
43	Array appears with inconsistent dimensions
44	Missing END statement or attempted execution of a nonexecutable statement
45	Array doubly dimensioned
46	Number of dimensions not obvious
47	Array too large
48	Out of storage during array allocation
49	Subscript too large
50	Accessed operand has undefined value
51	Noninteger power of negative number
53	Missing statement
55	RETURN finds no address
56	Out of data
57	Out of storage during execution
58	Dynamic array exceeds allocated storage
59	Dimensions not compatible
60	Matrix operand contains undefined element
61	Singular or nearly singular matrix
62	Trigonometric function argument too large
63	Attempted square root of negative argument
64	Attempted log of negative argument

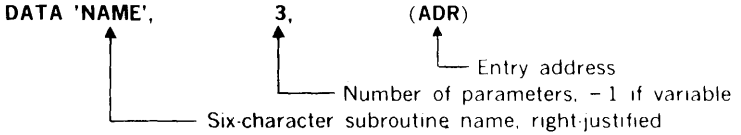
The following errors are warnings only; program execution continues:

65	Numerical overflow; result taken to be ± infinity
66	Numerical underflow; result taken to be zero
67	Log of zero taken to be − infinity
68	EXP overflow; result taken to be + infinity
69	Division by zero; result taken to be ± infinity
70	Zero raised to negative power; result taken to be + infinity

SECTION 6 – CALL DESIGN CONSIDERATIONS

There are certain internal design considerations for CALL subroutines. Parameters are passed to the called routine in the convention adopted by Varian; i.e., a list of the addresses of the actual parameters immediately following the JMPM instruction to the subroutine. All actual parameters are real (Varian floating-point format) numbers.

Subroutine linkage to the BASIC interpreter is through a list containing the name of the subroutine, the number of parameters it expects, and its entry address. The linkage list contains five words per subroutine, construction as follows:



The setting of the first word of the name field to zero signals the end of the list. A pointer to the linkage list must be placed in location 010. Location 022 points to the highest address used by BASIC, so this cell must be decremented to allow space for the linkage list and the user subroutines to be loaded above the BASIC tables.

A set of address bounds are available if the programmer wishes to know what type of parameter has been passed to him. By checking the parameters against these bounds, he can ensure, for example, that he does not store into constants. These address bounds are stored in the following location pairs (the first cell holds the first address of the block; the second holds the last address + 1):

Simple variable	$(011) \leq X < (012)$
Expression	$(015) \leq X < (0117)$
Constant	$(017) \leq X < (020)$

The following program shows a setup to make the two assembly language subroutines SUB1 and SUB2 available to a basic program.

To operate this program:

a. Load and initialize BASIC, selecting options as requested, until **READY** is typed.

b. Load the subroutine package containing SUB1 and SUB2 using the Binary Load/Dump program and overlaying locations 010 and 022.

c. Restart BASIC at location 02 to access the two subroutines.

```
SYMBOLS

1     015006 R  TABL
1     015003 R  SUB2
1     015000 R  SUB1
1     015000    LOC

*
*DEMONSTRATION OF TYPICAL SUBROUTINE ADDITION TO BASIC
*
          015000    LOC      ,EQU       ,015000
015000                       ,ORG       ,LOC       LOC OF SUBROUTINES
015000    000000    SUB1     ,ENTR      ,          ENTRY TO SUB1
                    *        .
                    *        BODY OF SUB1
                    *        .
015001    001000              ,JMP*      ,SUB1      RETURN FROM SUB1
015002    115000 R
015003    000000    SUB2     ,ENTR      ,          ENTRY TO SUB2
                    *        .
                    *        BODY OF SUB2
                    *        .
015004    001000              ,JMP*      ,SUB2      RETURN FROM SUB2
015005    115003 R
          015006 R  TABLE    ,EQU       ,*         LOC OF LINKAGE TABLE
015006    120240              ,DATA      ,'    SUB1',0,(SUB1) SUB1 ENTRY, ZERO PARA
015007    151725
015010    141261
```

```
015011    000000
015012    015000 R
015013    120322          ,DATA      ,    RESET',2,(SUB2) SUB2 ENTRY, TWO PARA
015014    142723
015015    142724
015016    000002
015017    015003 R
015020    000000          ,DATA      ,0        **END FLAG FOR LINK TABLE**
                  *
                  *NOW SET POINTERS FOR INTERPRETER
                  *
000010                    ,ORG       ,010      LOC OF INTERPRETER POINTERS
000010    015006 R        ,PZE       ,(TABL)   POINTER TO LINKAGE TABLE
000022                    ,ORG       ,022
000022    014777          ,PZE       ,(LOC-1)  POINTER TO LAST CELL
          000002          ,END       ,2
LITERALS

POINTERS

SYMBOLS

1     015006 R  TABL
1     015003 R  SUB2
1     015000 R  SUB1
1     015000    LOC
```

SECTION 7 - EXTENDED BASIC

Extended BASIC (EBASIC) extends the BASIC language to provide a powerful tool for real-time systems using rotating memory devices. A complete data-acquisition and process-control system called ADAPTS is built around EBASIC, and is available from Varian Data Machines.

This section describes how to write programs using the EBASIC language. The ADAPTS User's Guide (publication number 03-996 700B) contains additional programming information for the advanced user who wishes to add his own assembly-language subroutines to the system.

The two features of EBASIC which distinguish it from more common versions of BASIC are:

- A CALL statement which permits subroutines to be written in assembly language. Assembly language subroutines execute faster than subroutines written in EBASIC, and they also permit access to special purpose hardware.

- A set of statements which provide file handling and file maintenance capability. Programs and data files are resident on bulk storage devices such as fixed-head disc, moving-head disc, 9-track IBM compatible magnetic tape, or cassette magnetic tape.

GETTING STARTED

The best way to learn to program this system is to skim through this section very quickly and then to begin practice at the Teletype keyboard.

Assuming that EBASIC is running in the computer and the Teletype power switch is positioned to LINE, the computer will type READY each time the ESC key is used (CTRL, SHIFT and K for the ASR-35). In the ready state, the system is waiting for the user to type a control command or a numbered program statement.

Under normal operating conditions, EBASIC is always in either a ready state or a run state. If a program is in the computer, the system will enter the run state after the operator hits the control command RUN (followed by the carriage return key). This action will cause the program to be executed.

Keyboard Input

The Teletype keyboard is used to give control commands and to write program statements. From the standpoint of construction, the difference between a command and a program statement is that a program statement has a line number, for example,

```
10 PRINT X
```

and a command has no line number (or line number 0), for example,

```
PRINT X.
```

All lines of input must be terminated by the carriage RETURN key.

Control Commands

Several inputs are used specifically to operate on the entire EBASIC program in the computer. Therefore, these are designated as control commands. In fact, they may also be given line numbers and made part of a program. These commands, which are listed in table 7-1, must be terminated with the carriage RETURN key.

Table 7-1. EBASIC Control Commands

Command	Description
RUN nnnn	Runs or executes the program in the computer from line nnnn. If nnnn omitted, run begins at lowest numbered statement.
ESC key	Aborts program execution. Use CTRL, SHIFT, and K on ASR-35 Teletype.
LIST nnnn	Lists program from line nnnn to the end. If nnnn omitted, entire program is listed.
PLIST nnnn	Punches program from line nnnn to the end. If nnnn omitted, entire program is punched (paper tape output).
PTAPE	Reads in a program from paper tape. Each line is accepted or rejected exactly as if it were typed on keyboard.
SCRATCH	Deletes the entire program.
RESTART	Brings in a fresh copy of EBASIC from the system file (disc).
REMOVE mmmm TO nnnn	Removes program statements from line mmmm to nnnn. If nnnn≤ mmmm, then only one line will be removed.

The paper tape reader and punch may be either the Teletype devices or the optional high speed devices. The reader and punch are selected as Teletype or high-speed during the system dialogue with the user which follows the RESTART command. The RESTART command has a number of other functions as well:

- It enables the user to delete the matrix and/or trigonometric functions from EBASIC, thus conserving core space.
- It enables the user to load assembly language subroutines from paper tape onto the disc.
- It allows the user who has a system with more than 12K of core memory to make the upper core unavailable to EBASIC. That is, the upper core may be reserved, in 1K increments, for non-EBASIC use or for assembly language subroutines.
- It allows assembly language subroutines to be loaded from disc into core.

Program Statements

A description of each program statement is given in a later section. Each statement is preceded by a line number which must be an integer in the range from 1 to 9999. For example,

```
10 PRINT X
```

is a program statement. Statement numbers are assigned in the sequence in which the statements are to be interpreted at run time. However, they do not have to be typed in the order in which they are run. For example,

```
20 PRINT A
10 PRINT B
```

Even though statement 10 is typed after statement 20, it will precede 20 in the actual program. The LIST command types all program statements in their true numerical order - not the order in which they are entered.

Statements need not be executed in numerical order, however, because the program may contain branching statements which transfer control to another point in the program.

It is advisable to assign line numbers which follow in increments of 10 (or some other convenient number). This permits the insertion of statements between those previously entered.

Editing Features

Two keys on the Teletype have special meaning during input of a line.

← (black arrow)	• Deletes previous character on the line. Retype character or backspace again.
RUBOUT	• Abort current line. Must be given before RETURN to have effect. Retype line.

The back arrow may be used as many times as necessary on a single line to backspace over typographical errors. RUBOUT, in addition to aborting the entire line, will cause a backward slash (\) to be printed and the printer will advance to the left margin of the next line.

To change a program statement already entered, simply retype the line using the same line number. The new entry will overlay the old entry. To delete a single statement already entered, retype the line number and follow with the RETURN key. This "erases" the previous statement with that line number.

Error Messages

During the entry of a line, EBASIC checks for construction errors and reports certain errors. Errors may also be reported during running of a program. All errors are reported through the following message output to the user:

```
ERROR XX IN LINE nnnn
```

where nnnn is equal to line zero for control commands. The error codes (XX) and their meanings are given in table 7-2. Notice that errors 65 through 70 do not abort the program but serve only as warnings to the user.

ELEMENTARY BASIC

All subjects not included in this section may be ignored by the beginner, if he so desires. The omissions are those topics dealing with:

- arrays, matrices, and vectors
- bulk storage files

Assignment Statements

The topics covered in this section are:

- LET
- Variables and numbers
- Arithmetic and boolean operators
- EBASIC functions
- DEF

LET

The LET statement assigns a value to a variable. A simple example is:

```
10 LET A = 1
```

which assigns the value 1 to variable A. The symbol = is sometimes called a " replacement operator for" ; in this context, it means replace the value of the variable A with the constant 1.

The quantity to the right of the = symbol may be a formula involving previously defined variables and functions as well as constants. A simple formula is indicated by the example,

```
10 LET A = A + 1
```

in which the current value of A is replaced by A + 1, that is, A is incremented by one.

The word LET is optional and need not be typed by the user. Thus, the statement,

```
10 A = A + 1
```

has the same meaning as the previous example. LET is implied and understood by EBASIC. On a listing of the program, LET will be inserted before the variable to the left of the = symbol by the EBASIC language processor.

The LET statement may be used to assign one value to several constants. For example:

```
10 LET A = S = T = X = 0
```

The whole string of variables (A, S, T, X) will be assigned the value to the right of the last = symbol, which is zero in this case.

Variables

A variable may be either a simple variable or an array variable. The following variables are examples of each:

SIMPLE VARIABLES

A, B, ..., Y, Z
A0, A1, ..., B0, B1, ..., Z8 Z9

ARRAY VARIABLES

A(1), A(2), ...,B(1), B(2), ... Z(255)
A(1, 1), A(1, 2), ..., B(1, 1), B, (1, 2), ..., Z(255, 255)
A(I), B(I, J), C(I, J2)

A simple variable may be either a single letter (A to Z) or a single letter followed by a single digit (0-9). An array variable is a single letter followed by one or two subscripts enclosed in parentheses. Subscripts may be previously defined variables but no subscript may exceed 255. Subscripts may also be integers.

Table 7-2. EBASIC Error Codes

Code	Meaning
1	Statement ends unexpectedly.
2	Input exceeds 72 characters.
3	System command not recognized. (May be missing statement number.)
4	Missing or incorrect statement type.
5	Exponent of number is missing power.
6	Symbol following MAT not recognized.
7	LET statement has no store.
9	Missing or incorrect function identifier in DEF.
10	Missing parameter in DEF statement.
11	Missing assignment operator.
12	Missing THEN.
13	Missing or incorrect FOR variable.
14	Missing TO.
15	Incorrect STEP in FOR statement.
16	Called routine does not exist.
17	Wrong number of parameters in CALL statement.

(continued)

Table 7-2. EBASIC Error Codes (continued)

Code	Meaning
18	Missing or incorrect constant in DATA statement.
19	Missing or incorrect variable in READ or OPEN statement.
20	No closing quote for PRINT string.
21	Missing print delimiter or bad PRINT quantity.
22	Illegal word follows MAT.
23	Missing delimiter.
24	Improper matrix function.
25	No subscript where expected.
26	May not invert or transpose matrix into self.
27	Missing multiplication operator.
28	Improper matrix operator.
29	Matrix may not be both operand and result of matrix multiplication.
30	Missing left parenthesis.
31	Missing right parenthesis.
32	Operand not recognized.
33	Defined array missing subscript part.
34	Missing array identifier.
35	Missing or bad integer.
36	Nonblank characters following statement's logical end.
37	Out of storage during syntax phase.
38	Tape reader not ready or EOF on mag tape.
39	Doubly-defined function.
40	FOR statement has no matching NEXT statement.
41	NEXT statement has no matching FOR statement.
42	Formal parameter finds no actual parameter.
43	Array appears with inconsistent dimensions.
44	Missing END statement.
45	Array doubly dimensioned.
46	Number of dimensions not obvious.
47	Array too large.
48	Out of storage during array allocation.
49	Subscript exceeds bound.
50	Accessed operand has undefined value.
51	Noninteger power of negative number.
53	Missing statement.
55	RETURN finds no address.
56	Out of data.
57	Out of storage during execution.

Table 7-2. EBASIC Error Codes (continued)

Code	Meaning
58	Dynamic array exceeds allocated storage.
59	Dimensions not compatible.
60	Matrix operand contains undefined element.
61	Singular or nearly singular matrix.
62	Trigonometric function argument is too large.
63	Attempted square root of negative argument.
64	Attempted log of negative argument.
65	Numerical overflow, result taken to be ± infinity.
66	Numerical underflow, result taken to be zero.
67	Log of zero taken to be − infinity.
68	EXP overflows, result taken to be + infinity.
69	Division by zero, result taken to be ± infinity.
70	Zero raised to negative power, result taken to be + infinity.
71	Illegal file name.
72	Illegal file type.
73	Illegal logical file number.
74	File number not ASSIGNed.
75	Illegal file access mode.
76	File not OPENed
77	File cannot be found.
78	Mass storage full.
79	Attempt to read past end of file.
80	I/O unit not ready.
81	Data transfer I/O error.
82	Peripheral not in configuration.
83	Removable media peripheral busy.
84	No BASIC program in core to SAVE.
85	Illegal argument(s).
86	Incorrect number of arguments.
87	Data file too large.
88	System power on or run from location zero.

Note: Errors 65 through 70 are warnings only. Program will continue to execute.

Numbers

Numbers in EBASIC are positive or negative expressed in decimal form, and may contain up to seven significant digits. Numbers may be in the range 10^{-38} to 10^{+38}

Example:

```
-3.5
5
-7E-5
1234567
6E+10
1.543217E6
```

where -7E-5 means -7 times 10 raised to the -5 power. The plus sign is optional for a positive power of 10.

Numbers may be entered in one of three ways:

a. As an integer (no decimal point)

b. As a real number (with a decimal point)

c. As large number (integer or real) with an exponent to the base 10 (scientific notation). Maximum exponent is ± 38.

When printing numbers during a program run or following a LIST command, EBASIC uses a format which depends on the size and type (real or integer) of the number. This function is explained later under the PRINT statement. For now, assume that an integer of more than six digits will be printed in exponential notation as will very large or very small real numbers.

All numbers are stored in computer memory in a floating-point format that utilizes two 16-bit computer words per value. This is true whether the numbers are referenced as contants, simple variables, or array variables.

Arithmetic and Boolean Operators

The arithmetic and boolean operators enable the user to perform computations in EBASIC. Functions (described in following section) may also be used to perform computations. The operators and the operations they perform are shown in table 7-3. The order in which they are listed indicates their priority or the order in which the operations will be performed in a formula. Exponentiation is of the highest priority and the boolean OR of the lowest. Operations on the same line in the table are of equal priority.

Table 7-3. Arithmetic and Boolean Operators in Descending Order of Priority

Operation	Operator
Exponentiation	↑
Multiplication, division	*, /
Addition, subtraction, NOT	+, −, NOT
AND	AND
OR	OR

The following are examples, shown both in EBASIC notation (using the operators) and in notation perhaps more familiar.

EBASIC Notation	Algebraic Notation
`X ↑ 2 + 2`	$X^2 + 2$
`(A/(B+4)) *X`	$\left(\frac{A}{B+4}\right)X$

EBASIC Notation	Boolean Notation
`A AND B OR NOT C`	$A \cdot B + \overline{C}$

Boolean operators are defined by " truth tables" , which show the effects of the operations. The truth tables for AND, OR, and NOT are given in tables 7-4.

Table 7-4. Truth Tables for Boolean Functions AND, OR, NOT

AND			OR			NOT	
A	B	A AND B	A	B	A OR B	A	NOT A
F	F	0	F	F	0	F	T
F	T	0	F	T	1	T	F
T	F	0	T	F	1	--	--
T	T	1	T	T	1	--	--

Note: F = 0; T≠ 0

A value (B for example) is false (F) if B is equal to zero and true (T) if B is not equal to zero.

The rules of priority may be summarized as follows:

a. In the absence of parentheses, all arithmetic and boolean operations in a formula are computed in the priority indicated by table 7-3. Exponentiation has the highest priority and multiplication and division are next (equal priority).

b. A quantity in parentheses is computed before that quantity is used in further computation.

c. In the absence of parentheses in a formula involving operations of equal priority, they are performed as they are read from left to right.

Functions

In addition to the five arithmetic operations (↑, *, /, +, −) and the three boolean operations (AND, OR NOT), EBASIC has twelve defined functions. These functions are listed in table 7-5.

Table 7-5. EBASIC Functions

Function	Interpretation
SIN(X)	Find the sine of X
COS(X)	Find the cosine of X
TAN(X)	Find the tangent of X
ATN(X)	Find the arctangent of X
EXP(X)	Find e^X
LOG(X)	Find the natural logarithm of X (lnX)
SQR(X)	Find the square root of X ($\sqrt{X}$)
ABS(X)	Find the absolute value of X
INT(X)	Find the largest integer contained in X
RND(X)	Find a random number (uniform distribution) X is dummy
SGN(X)	SGN(X) = 1 if X > 0, −1 if X < 0, 0 if X = 0
TAB(X)	Space to column X, where X = 0 through 71

TRIG FUNCTIONS

The trigonometric functions (SIN, COS, TAN and ATN) may be deleted from the system by the user during the Teletype dialogue following the use of the RESTART command. The user may wish to delete these functions, since this will provide an extra 400_{10} locations in computer memory for use by EBASIC.

INT

The INT (integer) function may be used to round off to the next highest integer by adding 0.5 to the number before applying the function. For example:

```
10 LET A = 3.7
20 LET B = INT (A)
30 LET C = INT (A + 0.5)
```

In this example B would be assigned to the value 3 while C would be assigned the value 4.

RND

The RND (random) function is used to generate a random number from a uniform distribution of numbers between 0 and 1. The argument for RND has no meaning, although it must be included. That is,

```
RND (X)
RND (12)
RND (0)
```

are all equivalent. For example, to generate a random integer between 0 and 100, the INT function could be used as follows:

```
10 LET X = INT (100 * RND (0))
```

It is important to remember that when RND is used in a program, the same sequence of random numbers will be generated each time the program is run. If the user desires a different sequence he may discard some numbers from the sequence.

The following example generates random numbers from 0 to 100 and prints them until a number greater than 50 is generated. Upon finding a number greater than 50, the program ends and EBASIC returns to a ready state.

```
10 LET X = 100 * RND (0)
20 IF X > 50 THEN 100
30 PRINT X
40 GOTO 10
100 END
```

SGN

The SGN (sign)function returns the value – 1, 0, or + 1 according to whether the argument is negative, zero, or positive, respectively.

TAB

The TAB function is used in conjunction with the PRINT statement to format output. It may cause the printer to advance to the column (0-71) specified by the argument before printing. The TAB function should be followed by the item to be printed.

DEF

In addition to the standard functions, any other function which will be used in the program a number of times can be defined by the use of the DEF statement. The name of the defined function must be three letters, the first two of which must be FN. Thus, a total of 26 functions can be defined FNA, FNB, etc.).

For example, if the function,

$$\frac{\text{SIN (X)}}{\text{X}}$$

is needed frequently, it might be defined by the line

```
30 DEF FNR (X) = SIN (X)/X
```

The expression to the right of the equal sign may be any formula which fits onto one line. It may include standard functions and even functions defined by other DEF statements. The DEF statement may appear after the line number in which the function it defines is used, but not after and END statement.

The following sequence of statements illustrates the use of a function to compute the third side of a triangle when two sides and the included angle are given. Note that, in the definition, the two known sides (B1, C1)are given as variables. These variables must be assigned values before the function may be used. The variable A (the included angle) is used to define the function. Function FNS converts the angle given in degrees to its radian equivalent. In the example FNR computes the side of the triangle and assigns it equal to the variable S.

```
10 LET B1 = 10
20 LET C1 = 20
30 LET A  = 30
40 LET S  = FNR(A)
...
...
100 DEF FNR(A) = B1 ↑ 2 + C1 ↑ 2 - 2 * B1 * C1 * FNS(A)
110 DEF FNS(A) = (A * 3, 14159) / 180
```

Data Pools

This section describes the method of establishing and using data pools. A data pool is a collection of values to be used by the program during run time. The section shows how to establish a one-to-one correspondence between variables and constants. This permits a program to run according to input which the user may enter prior to execution.

READ DATA

The READ statement is always used in conjunction with one or more DATA statements. A DATA statement establishes a pool of numerical constants which may be assigned to variables in a READ statement. In the following example, X, Y, and Z are assigned values 1, 2 and 3 respectively.

```
10 READ X, Y, Z
20 DATA 1, 2, 3
```

The items following READ are called a READ list and the items following DATA are called a DATA list. The DATA list may contain constants only; it may not contain functions, expressions, or operators. DATA statements may appear anywhere in the program since they are not executed.

Access to the data pool by READ statements is always sequential in the order in which items appear. Constants placed in the data pool need not be used, but enough values must be supplied to satisfy all READ statements. Continuing the example already introduced,

```
10 READ X, Y, Z
20 DATA 1, 2, 3
30 DATA 4, 5, 6, 7, 8
40 READ A, B, C
```

in which A, B, and C are assigned the values 4, 5 and 6 respectively and the values 7 and 8 are not read at all.

RESTORE

The RESTORE statement permits the data pool to be re-used by initializing the READ pointer to the first value in the lowest numbered DATA statement. In the following example, X and Y are initially assigned the values 1 and 2, respectively. X and Y are then used in a LET statement which alters the value of X. After executing the RESTORE statement, the READ statement in line 50 re-uses the data pool, only this time the values are assigned in reverse order.

```
10 READ X, Y
20 DATA 1, 2
30 LET X = X + Y
40 RESTORE
50 READ Y, X
...
...
```

Miscellaneous Statements

This miscellaneous statements are presented here under the assumption that the beginner may be reading this chapter in the order of the section numbers. At least two of these statements discussed will be used frequently in examples in later sections.

END

Every program must have an END statement. Its form is simple, a line number followed by END, for example,

```
1000 END
```

Any number of END statements are permitted. Execution of an END statement causes EBASIC to return to the ready state, in which the system awaits commands from the keyboard.

Example:

```
10 READ X
200 DATA 1000
300 END
```

Note that the END statement need not be the highest numbered statement in the program. However, the END statement must be placed somewhere in the program following all DEF statements and DATA statements.

STOP

The STOP statement may be used interchangeably with the END statement to return the system to a ready state. STOP and END have identical meaning in EBASIC.

REM

REM provides a means for inserting explanatory remarks in a program by instructing the computer to ignore the remainder of the line. This allows the user to follow REM with directions for the use of the program, identification of parts of a long program, or anything else he wants. Although the remarks following REM are ignored, the line number of a REM statement may be used in a GOTO or IF THEN statement. A line number of a REM statement may also be used in a GOSUB statement. Sample REM statements are:

```
10 REM  THIS IS A REMARK.
20 REM  ALTHOUGH THE COMMENTS FOLLOWING REM
30 REM  ARE IGNORED BY BASIC, IT
40 REM  TAKES TIME TO DETECT THE PRESENCE
50 REM  OF A REM IN THE PROGRAM.  THEREFORE,
60 REM  AVOID USING REM IN A LOOP.
```

WAIT

The WAIT statement introduces a delay into the program. Execution of the WAIT statement causes the program to wait for a specified number of milliseconds. An example of a WAIT statement in a program is

```
30 WAIT  100
```

Because a software timer is used by WAIT, the program delay depends upon the speed by which the instructions are processed. Program delays are listed below for the Varian computers:

COMPUTER	DELAY (in milliseconds)
620/L	100
620/L-100	50
620/f	42
620/f-100	40
V73	30

In the 620/L-100, 620/f-100, and V73 computers, precision timing can be programmed using the hardware real-time clock (see ADAPTS User's Guide).

The argument following WAIT may be a constant, a variable, or a formula. The argument is truncated to an integer and converted to a 16-bit computer word before it is used. Therefore, the argument must evaluate to a value not greater than 32767 and not less than zero. Negative values will result in an error message being issued to the user. Positive values greater than 32767 will be treated as 32767 and will not generate an error message.

If in range, the value is decremented every millisecond on the 620/L and once every 0.40 millisecond on the 620/f-100. When the value reaches zero, the delay period is over.

The following use of a function, FNT, enables the WAIT statement on a 620/f-100 simulate the timing on a 620/L.

```
10 DEF FNT (T) = T/.40
20 READ T
30 DATA 1000
40 WAIT FNT (T)
.....
....
```

Branching Statements

A branching statement alters the sequence in which program statements are encountered during the running of a program. Two types of branching statements are presented here:

a. The unconditional branch, represented by GOTO and GOTO OF

b. The conditional branch, represented by IF THEN

The GOTO OF and IF THEN statements permit a program to make decisions based upon input or computations.

A third type of branch, a subroutine branch, is considered as a separate topic in a later section.

GOTO

The statement form is GOTO line number. It is possible to go to a non-executable statement; in this case control passes to the next executable statement in sequence.

An example of the GOTO statement is:

```
150 GOTO 75
```

GOTO may be typed as one word or two separate words. Imbedded spaces are ignored. In a listing printed by BASIC, GOTO will be a single word.

GOTO OF

The GOTO OF statement is of the form

GOTO formula OF list of line numbers

The formula is evaluated and truncated to an integer N, which selects the Nth line number in the list as the target for the GOTO. Error messages are reported if:

a. The formula evaluates to less than one or to a number greater than the number of lines in the list; or

b. The selected line number for the GOTO is nonexistent.

Example:

```
10 LET I = 3.3
20 GOTO I-1 OF 200, 300, 400
......
200 REM TARGET IF INT (I-1) = 1
.......
300 REM TARGET IF INT (I-1) = 2
......
400 REM TARGET IF INT (I-1) = 3
```

IF THEN

The IF THEN statement is of the form

IF formula relation THEN line number

For example:

```
10 IF A = 1 THEN 200
20 REM TARGET IN A # 1
.....
200 REM TARGET IF A = 1
```

If the relation following IF is satisfied, control will be directed to the line number following THEN; if the relation is not satisfied, control is passed to the next statement in sequence. A full list of the relational operators follows:

Relational Operator	Definition
=	Equal To
<	Less Than
>	Greater Than
<=	Less than or equal to
>=	Greater than or equal to
#	Not equal to

Input/Output Statements

This section describes the INPUT statement, the PRINT statement, and the use of the TAB function with PRINT. The INPUT statement allows the operator to enter data from the keyboard during the running of a program. Print allows the program to print and format alphanumeric output from the program. The use of the TAB function gives PRINT additional format flexibility.

INPUT

There are times when it is desirable to enter data during the running of a program. A data entry request during run may be accomplished by an INPUT statement, which acts as a READ statement but does not draw data from a DATA statement. For example, if the EBASIC program requires the user to supply a value for X during the program run. The following statement may be coded

```
40 INPUT X
```

before the first statement to use the value of X. When the computer encounters this statement at run time, it prints a question mark and waits for the user to enter a number;

it will not accept letters, functions, or expressions. After the user types a number and presses the carriage RETURN key, control passes to the next statement in sequence.

Frequently a PRINT statement is employed to tell the user what response to make when he sees the question mark:

```
10 PRINT 'ENTER DESIRED VALUE FOR X'
20 INPUT X
```

At run time, the program will print

```
ENTER DESIRED VALUE FOR X
?
```

and wait until it receives a number followed by a carriage RETURN.

Multiple values may be requested by a single INPUT statement (as many as fit on a line). For example:

```
100 INPUT X, Y, Z
```

In this case, when the question mark is printed by the program, the user may enter the three values in two ways:

a. He may type all values on the same line, separated by commas and followed by carriage RETURN.

b. He may strike carriage RETURN after typing each value. The program will continue to print a question mark on each line until all values have been entered.

PRINT

The PRINT statement may be used to print and format alphanumeric output. The following examples illustrate PRINT statement usage:

`10 PRINT 'MESSAGE'`	The message in quotation marks is printed exactly as typed.
`20 PRINT X`	Print the value of X
`30 PRINT 3.5`	Print the number 3.5
`40 PRINT 2*SIN(X) + 5`	Print the value computed by the formula
`50 PRINT`	Print a blank line. Used to space text

In all of the above examples, except in line 50, PRINT is followed by a single item which is a message, a variable, a constant, or a formula. Each item will be typed on a single line. A PRINT list may contain as many items as can be typed on a line; items are separated either by commas or semicolons. The list itself may be terminated by a comma or a

semicolons. The list itself may be terminated by a comma or a semicolon. Commas and semicolons provide output formatting capability as described in the following paragraphs.

A line of output is divided into 72 columns labeled 0 through 71. The first item in a PRINT list will normally be printed starting in column 0. The line is arbitrarily divided into five zones which start in columns 0, 15, 30, 45, and 60. A comma following an item in the list is a signal to advance to the next zone. If a semicolon follows an item in the list, no extra spaces are inserted before printing the next item.

If a comma terminates a list in a single PRINT statement, it effectively links the list in the next PRINT statement. For example,

```
10 PRINT X, Y,
20 PRINT Z
```

is equivalent to

```
10 PRINT X, Y, Z
```

When past the beginning of zone 5 (column 60) the comma is the signal to begin in column 0 on the next line.

A semicolon at the end of a list suppresses the carriage return. Therefore, it may be used to link an output message to an INPUT request as follows:

```
10 PRINT 'ENTER VALUE FOR X';
20 INPUT X
```

When the program is run, the computer will print

```
ENTER VALUE FOR X?
```

whereupon the user types in a number following the question mark. If the semicolon after the print message were removed, then the question mark would have been printed on the next line.

The PRINT statement may be used without a line number to give immediate results of a computation. For example,

```
PRINT 2*3
6
```

where 6 is printed by the computer on the next line after operator input of the PRINT statement.

PRINT may be used to debug programs when an error causes a program to abort. All variables after a normal or aborted run will contain the last values used in the program. For example, if the program has ended or aborted and the user types

```
PRINT N
```

the value last assigned to N will be printed on the next line.

FIELD DEFINITIONS FOR NUMBERS

When a number is output from a PRINT statement, it is printed in a format which depends on the size of the number and whether it is an integer. This format is called a field definition. The number of spaces (Teletype columns) is called the field. The field definitions for EBASIC numbers are given in table 7-6. Notice that all fields include trailing blanks (spaces) so that numbers printed in adjacent fields are easily readable.

A simple PRINT statement like

```
10 PRINT 37; 9998
```

will normally cause the field on which the number 37 is printed to begin at column zero. According to table 7-6, the next number (9998) will begin in column 6, because a semicolon has been used after the 37. If a comma had been used to separate 37 and 9998, then 9998 would have begun its field in column 15.

Table 7-6. Field Definitions for Values Printed by EBASIC

Value of Number	Type of Number	Field Definition
$1 \leq /n/ \leq 999$	Integer	SXXXbb
$1000 \leq /n/ \leq 999999$	Integer	SXXXXXXbbbb
$0.1 \leq n \leq 9999.9$	Real (normal range)	SXXXXXXbbb
$n < 0.1$	Small or real integer	SX.XXXXXE ± eebb
$n > 9999.9$	Large real	SX.XXXXXE ± eebb
$n > 999999$	Large integer	SX.XXXXXE ± eebb

Notes:

1. Each X represents a decimal digit (0-9) except that trailing zeros are replaced by blank spaces for integers and real numbers in the normal range. Numbers are left justified in the field.

2. S is equal to the minus sign (−) for negative numbers and is a blank space for positive numbers.

3. b indicates a blank space.

4. For real numbers in the normal range, one of the X's is replaced by a decimal point.

(continued)

5. Numbers in the extreme range are printed in scientific notation, where ee represents the exponent to base 10. The value of ee cannot exceed 38.

6. Internal representation of numbers is always in a floating point format, which requires two 16-bit computer words for each number.

USE OF TAB WITH PRINT

The TAB function may be used to advance the printer to the specified column before printing. TAB will have no effect if that column has already been passed or if the TAB argument exceeds 71 (the last column). The TAB function is given immediately before the item to be output. For example

```
10 PRINT TAB (32) X, TAB (65) Y
```

will print the value of X beginning at column 32 and Y beginning at column 65. If, however, the following statement is used:

```
10 PRINT TAB (32), X, TAB (65) Y
```

we find that X now begins in column 45. Tab (32) advances to column 32 but the comma before X advances printing to the next zone, which begins at column 45. A semicolon between TAB (32) and X would have no effect, however.

Writing Loops

Frequently, it is necessary to write a program in which one or more portions are performed not just once but a number of times, perhaps with slight changes each time. The programming device known as a loop is used to perform this iterative processing.

One type of loop is illustrated by the following example:

```
10 LET A = 1
20 GOTO 10
```

This is not a very useful loop, nor is there any way to terminate it except through the use of the ESC key. In order to execute a loop a finite number of times, the program must be provided with decision making capability. One way to do this is through the use of the IF THEN statement, which has already been introduced.

For example:

```
10 LET A = 0
20 A = A + 1
30 IF A > 10 THEN 50
40 GOTO 20
50 END
```

This loop consists of lines 20, 30 and 40 and is executed 10 times before the program ends. Another way of writing the program to perform the same function is:

```
10 LET A = 0
20 A = A + 1
30 IF A ≤ 10 THEN 20
40 END
```

This loop uses one line less than the previous example and may, therefore, be termed more efficient.

Another way of writing loops is to use the FOR and NEXT statements. FOR and NEXT are always used together. They mark the boundaries for a FOR-NEXT loop. For example:

```
 LIST
10 FOR I = 1 TO 10 STEP 5
20 PRINT I, SQR (I)
30 NEXT I
40 END
  RUN
  1          1
  6          2.44948
READY
```

The first time through the loop, I (a running variable) is set equal to the initial value (in this case, 1) and a test is made to see if I exceeds the final value (in this case, 10). Since in this example, the initial value does not exceed the final value, control passes to the PRINT statement. If the initial value were greater than the final value (less than, for negative step size) control would pass immediately to the line number following NEXT, which is line 40 in this example.

The main body of the loop is included between FOR and NEXT. When the program gets to the NEXT statement, it increments the running variable (I) by the step size and tests it. If the running variable has not passed the boundary set by the final value, control passes to the first statement after FOR. Otherwise, control passes to the statement following NEXT.

The initial value, the final value, and the step size may all be formulae of any complexity which can be typed on a single line. Initial value, final value, and step size may also evaluate to negative or positive numbers or zero.

The NEXT statement must include the running variable (in this case, I). NEXT increments the running variable by the step size but this does not prevent statements within the loop from also operating on the running variable to change its value. For a positive step size the loop continues as long as the running variable is less than or equal to the final value. For a negative step size, the loop continues as long as the running variable is greater than or equal to the final value.

The step size may be omitted from the FOR statement if a step size of plus one is to be used. For example:

```
 LIST
10 FOR I = 1 TO 10
20 PRINT I, SQR (I)
30 NEXT I
40 END
 RUN
 1          1
 2          1.41421
 3          1.73205
 4          2
 5          2.23606
 6          2.44948
 7          2.64575
 8          2.82842
 9          3
 10         3.16227
 READY
```

The main body of a FOR-NEXT loop may contain an IF THEN statement which causes the program to exit from the loop before the running variable reaches its final value. The body may also contain GOSUB, GOSUB OF, or GOTO statements which cause the program to exit from the loop either temporarily or permanently. The body may even contain other FOR-NEXT loops. These are called nested FOR-NEXT loops. However, they must actually be nested and must not cross, as illustrated below.

Allowed

```
FOR X
  FOR Y
  NEXT Y
NEXT X
```

Not Allowed

```
FOR X
  FOR Y
NEXT X
  NEXT Y
```

Allowed

```
FOR X
  FOR Y
    FOR Z
    NEXT Z
    FOR W
    NEXT W
  NEXT Y
  FOR Z
  NEXT Z
NEXT X
```

Subroutines

There are two types of subroutines which may be used in EBASIC. These are (1) subroutines written in the EBASIC language; and (2) subroutines written in assembly language. They are usually written for different purposes. EBASIC subroutines are written

to perform operations which may be useful several times in a program or at different parts of a program. Assembly language routines are usually written to gain access to special purpose hardware. For example, assembly language subroutines are used in this system to access the Interface Console.

Assembly language routines add a degree of flexibility and control to the system, but they may be run only on the specific computer for which they are written. Subroutines written in EBASIC are in a high-level language which has been implemented on a number of different computers.

The EBASIC statements used in writing and accessing EBASIC subroutines are:

- GOSUB
- RETURN
- SUB
- GOSUB OF

A single EBASIC statement, the CALL statement, is used to access any and all assembly-language routines which have been written in the proper format.

GOSUB-RETURN

The GOSUB statement is of the form:

GOSUB line number

For example,

```
75 GOSUB 210
```

When the GOSUB is executed, control passes unconditionally to the line number referenced. The last statement executed in a subroutine must be RETURN, which transfers control to the next statement after the GOSUB.

The following example shows the use of a subroutine to calculate the sine of an angle A given in degrees. The routine uses a Taylor series expansion. Compare the value produced by the subroutine to the true value (1/2) and the value given by the SIN function in EBASIC (.499999). The computed value of .499999 is close enough for most purposes. The RETURN statement is executed when the last term used in the series approximation is less than or equal to 10^{-6} . From the mathematical properties of the series we know that this last term is also less than the error introduced by terminating the series at this point. However, the calculated value shows that roundoff errors have reduced the accuracy.

A subroutine must be terminated by the use of a RETURN statement. Statements such as IF THEN and GOTO are illegal. More than one RETURN statement is permitted inside a subroutine but one is sufficient.

A GOSUB or GOSUB OF (see next section) may appear inside a subroutine. This procedure is known as " nested GOSUB's" .

Notice that all variables used inside the subroutine are " global" . This means that they have the same definition (or value) inside the subroutine as they do outside the subroutine. " Local" variables may be introduced by the use of a SUB statement (described in a later section) in the subroutine.

```
 LIST
10   LET A = 30
20   GOSUB 50
30   PRINT 'SINE OF'; A; 'IS'; S
40   END
50   REM SUBROUTINE TO CALCULATE SINE OF A
60   LET X = 3.14159*A/ 180
70   LET S = 0
80   LET N = N2 = Z = 1
90   LET X2 = X
100  GOTO 150
110  LET N2 = N + 2
120  LET N2 = N* (N - 1) *N2
130  LET X2 = X2*X↑2
140  LET Z = -Z
150  LET I = Z*X2/N2
160  LET S = S + I
170  LET I = I*SGN (I)
180  IF I >1.00000E-06 THEN 110
190  RETURN
 RUN
SINE OF 30 IS .499999
READY
 PRINT SIN (3.14159*A/180)
 .499999
```

GOSUB OF

The GOSUB OF statement has the form

GOSUB formula OF list of line numbers

where the value of the formula is truncated in an integer, N, which selects the Nth line number in the list. If N is less than one or greater than the number of line numbers in the list or if the target line number is non-existent, then an error is reported.

Note that GOSUB OF may not pass parameters to a subroutine. A subroutine entered by GOSUB OF must be terminated by execution of a RETURN statement.

SUB

The SUB statement is used to define parameters which are passed to a subroutine. A SUB statement may look like the following

```
300 SUB N, A, B (2), C
```

and the corresponding GOSUB may be something like

```
80 GOSUB 300, X, 5, Y, Q(1)
```

The SUB statement picks up the argument list following the line numbers in the GOSUB statement and assigns their values to the list following SUB. Let us suppose that the following values are assigned to the GOSUB arguments:

```
X    = 3.5
Y    = 2
Q(1) = 11
```

then the SUB list will make the assignment

```
N    = 3.5
A    = 5
B(2) = 2
C    = 11
```

All arguments in a SUB list are " local" , that is, defined only within the body of the subroutine. If variables of the same name have been defined in the main program, they will not be affected by any operation which changes their values in the subroutine.

However, variable names used in the subroutine which are not in the SUB list are " global" . They are defined identically in the subroutine and in the main program.

The following example illustrates the method of parameter transfer between a GOSUB argument list and a SUB argument list. Line 60 was written as a deliberate error to show that Z, used in subroutine, is undefined in the main program. Z is a local variable. A, on the other hand, is a " global" variable, since it is not used in the SUB list.

Note the caution in lines 120 and 130. Z is equivalent to 10 and vice versa. Changing Z would change 10 and after the change every place the constant 10 is used in the program (except in quotation marks and in REM statements) the new value for Z would appear. If this happens inadvertently, simply retype the subroutine so that Z is reassigned to its initial value before running the subroutine. In most cases, the program may then be re-

run and the constant will be changed back to its original value. In some cases, however, it may be necessary to reload the program into the computer.

```
 LIST
10   READ I,K
20   DATA 3, 6
30   GOSUB 80, I, K, 10
40   PRINT 'I='; I; 'K='; K
50   PRINT 'A='; A
60   PRINT 'Z='; Z
70   END
80   REM SUBROUTINE BEGINS HERE
90   SUB X, Y, Z
100 LET X=X+Y
110 LET Y=Y+Z
120 REM CAUTION: DO NOT CHANGE Z IN SUBROUTINE. IF YOU DO SO,
130 REM THEN THE NEW VALUE FOR Z WILL BE STORED IN THE CONSTANT 10.
140 LET A=1
150 RETURN
 RUN
I=9   K=16
A=1
Z=
ERROR 50 IN LINE 60
READY
```

A subroutine which has a SUB statement may be entered by a GOSUB statement within the subroutine, passing new values to the SUB list. This is known as recursive entry and is illustrated by the following example. The subroutine is actually executed twice because after the first time through X is equal to 9, but on the second time through X is equal to 25. Therefore, the IF THEN statement (line 120) transfers control back to the main program.

```
 LIST
10    READ I,K
20    DATA 3, 6
30    GOSUB 60, I,K, 10
40    PRINT 'I=';'K=';K
50    END
60    REM SUBROUTINE BEGINS HERE
70    SUB X, Y, Z
80    LET X=X+Y
90    LET Y=Y+Z
100   REM RECURSIVE ENTRY INTO SUBROUTINE PERMITTED
110   IF X > 10 THEN 130
120   GOSUB 60, X, Y, Z
130   RETURN
 RUN
I=25   K=26
READY
```

ARRAYS

An array is an orderly presentation of numbers. The two primary ways of displaying numbers are by row and by column. An array which has one column only or one row only is said to be one-dimensional; if it has multiple rows and columns, it is a two dimensional array.

Array variables in EBASIC are identified by a single letter of the alphabet followed by one or two values enclosed in parentheses, for example,

A(1), B(2,3), C(I, J + K)

The value in parentheses indicates the subscripts that would be used in ordinary algebraic notation, for example,

A , B , C
1 2,3 I, J+K

where the first subscript identifies the row and the second subscript identifies the column of the array element. Table 7-7 illustrates the positions of the various elements in an array of four rows by three columns. By convention in Varian's EBASIC, a singly-subscripted array is designated as a column vector (as opposed to a row vector).

Table 7-7. Array Element Positioning and Method of Storing in Computer Memory

Row \ Column	1	2	3
1	A(1, 1)	A(1, 2)	A(1, 3)
2	A(2, 1)	A(2, 2)	A(2, 3)
3	A(3, 1)	A(3, 2)	A(3, 3)
4	A(4, 1)	A(4, 2)	A(4, 3)

Note: Array elements are stored by column in computer memory. A(2, 1) follows A(1, 1). . . A(1, 2) follows A(4, 1), and so on. DATAI and DATAO (assembly language subroutines) access array elements according to this storage scheme.

Array Subscripts

Array subscripts may be constants, simple variables, array variables, or formulae. All subscripts are truncated to an integer before they are used. Subscripts must evaluate to a number from 1 to 255. An error will be reported at run time if sufficient storage area is not available in the computer for the array element referenced (see next section). The following example shows the use of a simple variable and an array variable as subscripts.

```
 LIST
10   DIM A (10), B (10)
20   FOR I=1 TO 10
30   LET A (I) =I+ .2
40   LET B(A(I)+ .1) =SQR (I)
50   PRINT A (I),B(I)
60   NEXT I
70   END
RUN
 1.2          1
 2.2          1.41421
 3.19999      1.73205
 4.2          2
 5.19999      2.23606
 6.19999      2.44948
 7.19999      2.64575
 8.19999      2.82842
 9.2          3
 10.1999      3.16227
READY
```

DIM Statement

The dimensions of arrays in EBASIC should be declared in the program. This is done through the use of a DIM statement.

Example:

```
25 DIM A (5), B(20,30), C(20), D(1,10)
```

In this example A is declared to be one-dimensional array with 5 elements. B has 20 rows and 30 columns, C has 20 elements (in a column, like A) and D has 1 row and 10 columns. Array D is a degenerate case of a doubly-dimensioned array since one of the dimensions is 1. However, this is the only way we have of producing an array of one row or a row vector in BASIC (arrays A and C are columns).

RULES FOR DIM STATEMENT:

a. A DIM statement reserves storage in computer memory for the arrays in the list following DIM. Arrays follow one another in memory in the reverse order in which they appear in DIM statements.

(continued)

b. The values for the dimensions enclosed in parentheses in a DIM statement must be constants in the range 1 to 255. If any dimension exceeds 255, no storage space will be reserved and no error will be reported on statement entry. An error will be reported, however, on any attempt to use the incorrectly dimensioned array in the program.

c. Array dimensions do not have to be declared in a DIM statement if no array variable in the program uses a subscript greater than 10. Although this is a convenient feature, the use of array variables without DIM statements is to be discouraged because, if dimensions are not declared, BASIC will automatically reserve storage for 10 elements in the case of an array variable with single subscript and will reserve storage for 100 elements for an array variable with two subscripts. This is very wasteful if all the space reserved is not really needed.

d. DIM statements may appear anywhere in the program, since they are not executed. It is good practice, however, to group them at the beginning of the program where they are easily identified.

Matrix Statements

A matrix is a two-dimensional array which is subject to certain rules of operation or manipulation. The explanation of the matrix statements in EBASIC assumes that the user understands these rules, which are readily found in any elementary textbook on matrix algebra.

The matrix statements, which are listed in table 7-8, may be deleted from EBASIC during the Teletype dialogue which follows the use of the RESTART command. Deleting the matrix statements frees approximately 800 words of computer memory for use by the program.

Normally, the dimensions of all matrices used must be dimensioned with DIM statements. If the array is to contain a number of elements not to exceed 100, EBASIC offers the convenience of using the MATREAD, ZER, CON, and IDN commands to specify the number of elements, as in the following examples.

```
10 MATREAD A(3,7)
20 MAT B = ZER (2, 10)
30 MAT C = CON (20,5)
40 MAT D = IDN (10,10)
```

CAUTION

Although it is possible to use the MATREAD, ZER, CON, and IDN commands to assign the elements in a matrix, the use of dimensions in these commands is discouraged for the following reasons:

a. If no DIM statement appears in the program to allocate core space for the array, EBASIC will reserve storage for 100 elements, even if fewer elements are required.

(continued)

b. ZER, CON, and IDN will have the effect of decreasing the core memory allocation in a non-reversible manner if the dimensions given are less than those in a DIM statement which refers to the same array. However, if one of these commands attempts to use more core than is alloted, an error will be reported.

For the reasons listed above, it is hoped that the programmer will use dimension specifications only in DIM statements. There is one case, however, in which he may wish to go against this advice; this is the case in which the number of elements to be assigned is unknown but certain to be less than 100. In such cases, the dimensions in a MATREAD, ZER, CON, or IDN may be given as variables or formula. This is not true for the DIM statement which uses only constants to specify dimensions.

Table 7-8. EBASIC Matrix Commands

Command	Meaning
MATREAD A, B, C	Read the three matrices. Data is stored row by row (not column by column).
MAT C = ZER	Fill out matrix C with zeros.
MAT C = CON	Fill out matrix C with ones.
MAT C = IDN	Set up C as an identity matrix.
MATPRINT A, B;C	Print the three matrices with A and C in regular format and B in closely packed format.
MAT B = A	Set matrix B equal to matrix A.
MAT C = A+B	Add the two matrices A and B.
MAT C = A-B	Subtract matrix B from matrix A.
MAT C = A*B	Multiply matrix A by matrix B.
MAT C = TRN(A)	Transpose matrix A.
MAT C = INV (A)	Invert the matrix A.
MAT C = (K) * A	Multiply the matrix A by the number K. This is multiplication of a matrix by a scalar.

Use of Matrix Operations

Certain operations are legal or illegal according to the standard rules which govern matrix algebra. For example, if matrix A has dimensions L by M (L rows and M columns) and matrix B has dimensions M by N, then

A * B is legal (resulting dimension: L by N)

whereas

B * A is illegal

Restrictions

There are several restrictions on the use of matrix commands on the Varian system which are imposed because of the necessity to use computer memory efficiently. Note that some of the legal examples below may be illegal in the sense that they violate principals of dimensioning.

a. Only one operation may be performed in a single statement.

Legal	Illegal
`MAT C = A+B`	`MAT C = A+B-D`

b. The results of multiplying one matrix by another matrix may not be stored in either, the multiplicand or the multiplier. In the second legal example K is a number rather than a matrix.

Legal	Illegal

`MAT C = A*B`
`MAT C = (K) * A`
`MAT A = (K)*A`
	`MAT A = A * B`

c. The transpose of a matrix or the inverse of a matrix may not be stored in itself. A violation of either of these rules is reported as error 26.

Legal	Illegal
`MAT B = TRN (A)` `MAT B = INV (A)`	`MAT A = TRN (A)` `MAT A = INV (A)`

MATREAD Statement

The MATREAD statement assigns constants from one or more DATA statements to the array variables. The array elements are assigned sequential values row by row.

For example:

```
 LIST
10   DIM A(3, 4)
20   MATREAD A
30   DATA 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
40   MATPRINT A
50   END
 RUN
 1             2             3             4

 5             6             7             8

 9             10            11            12
READY
```

MATPRINT Statement

The MATPRINT statements print one or more matrices as a result of a single statement. Elements are printed, row by row, either in the five zones across the page (beginning at columns 0, 15, 30, 45 and 60) or in closely packed format. As with the PRINT statement, the format is selected by following the array name with either a comma (or blank in the case of last array in the list) or a semicolon.

Examples:

MATPRINT A MATPRINT A, MATPRINT A,B	• Zone format in all three cases
MATPRINT A, B;	• Zone format for A and close packed format for B
MATPRINT A;B;	• Close packed format for A and B

All rows are double spaced. If a row overruns one line, it is continued on the next line (single spaced) until it is completely printed. An example of MATPRINT usage is:

```
 LIST
10   DIM A(2, 3), B(2, 2) C(2, 4)
20   MATREAD A
30   DATA 1, 2, 3, 4, 5, 6
40   MAT B=IDN
50   MAT C=CON
60   MATPRINT A,B;C;
70   END
 RUN
 1              2             3

 4              5             6

 1       0

 0       1

 1       1     1     1

 1       1     1     1
READY
```

VECTORS

A vector is defined as an array of one column or an array of one row. Thus a vector is either a column vector or a row vector. The matrix statements may be used on vectors as well as on arrays of two dimensions. In fact, the row vector is a degenerate case of a doubly-dimensioned array. For example,

```
10 DIM A(4), B(1,4)
```

defines B as a row vector whereas A is column vector. The following examples illustrate the use of the matrix statements with vectors. Note that in these examples, double spaced printing is not shown to conserve space.

```
 LIST
10   DIM A(4), B(1,4)
20   MATREAD A,B
30   DATA 1, 2, 3, 4, 11, 12, 13, 14
40   PRINT 'COLUMN VECTOR'
50   MATPRINT A
60   PRINT 'ROW VECTOR'
70   MATPRINT B;
```

(continued)

```
80  END
 RUN
COLUMN VECTOR
 1
 2
 3
 4
ROW VECTOR
 11      12      13      14
READY

 LIST
10  DIM A(4), B(1,4), C(4,4)
20  MATREAD A,B
30  DATA 1, 2, 3, 4, 11, 12, 13, 14
40  PRINT 'COLUMN VECTOR'
50  MATPRINT A
60  PRINT 'ROW VECTOR'
70  MATPRINT B;
80  MAT C=A*B
85  PRINT 'PRODUCT OF COLUMN VECTOR TIMES ROW VECTOR'
90  MATPRINT C;
100  END
 RUN
COLUMN VECTOR
 1
 2
 3
 4
ROW VECTOR
 11      12      13    14
PRODUCT OF COLUMN VECTOR TIMES ROW VECTOR
 11      12      13    14
 22      24      26    28
 33      36      39    42
 44      48      52    56
 READY
```

BULK STORAGE FILE HANDLING

All systems have at least one bulk storage file device, which is fixed- or moving-head disc. This is designated as the system file unit. The system file unit contains a copy of EBASIC and may also store program files, data files, and assembly language subroutines.

The system may be optionally equipped with one or two bulk storage devices which have removable file media. These are designated as file units A and B. Both A and B devices

may be either 9-track magnetic tape units (25 ips or 37.5 ips, 800 bpi), cassette tape units with Phillips cartridges, or a moving-head disc (File A only). The magnetic tape units can read and write in IBM compatible format.

Table 7-9 summarizes the file handling commands and the types of files on which they may be used. The file types are defined as follows:

EBASIC Program Files - These files, type P, are stored as ASCII characters; each item consists of four characters. They represent the statements of a complete EBASIC program.

Assembler Subroutine Files - These files, type S, are stored as relocatable object code of DASMAR assemblies. Each item is stored as a 16-bit word.

Floating Point Data Files - These files, type D, are stored as floating point data. Each item is a two-word floating point number.

Integer Data Files - These files, type I, are stored as integer data files, one integer data word per item. Type I files may be created only by use of the DATIF subroutine (described in a later section).

Loading EBASIC From the System File (RESTART)

The RESTART control command loads a fresh copy of EBASIC into the computer memory from the system file unit. After EBASIC is in memory following the use of RESTART, it enters into Teletype dialogue with the user, allowing him to select certain options. In addition to giving the RESTART command on the Teletype, the restart phase of EBASIC may also be entered (1) by manually entering and executing the restart bootstrap and (2) automatically, following the system generation phase.

Table 7-9. File Command Summary

Command	File Type			
FLIST (null,A,B)	P	S	D	I
SAVE name (null,A,B)	P			
LOAD name (null,A,B)	P			
COPY (name) TO (name)	P	S	D	I
DELETE name (null,A,B)	P	S	D	I
CLEAR (A,B)	P	S	D	I
ASSIGN name = file no.			D	I
OPEN file no., r/w, variable			D	I
CLOSE file no., variable			D	I
PUT file no., variable			D	I
GET file no., variable			D	I

The following example shows a sample of the Teletype dialogue which occurs during the restart phase. User responses are underlined. Each response is followed by the carriage RETURN.

```
TTY = 0, HSPT = 1
INPUT? 1                Select high-speed paper tape reader
OUTPUT? 0               Select Teletype punch
TRIG+MAT=2, TRIG=1,
NEITHER =0? 1           Select trig functions, omit matrix
                        statements*
CORE (K)?   15          Use lowest 15K of memory
APPEND LIBRARY? Y       Add Assembly language subroutines
APPEND LIBRARY? N       Done with additions
LOAD SUBROUTINES? Y     Load assembly language subroutines
                        into memory
NAME? 'IOSS'            Get IOSS from disc
NAME? 'CRT'             Get CRT from disc
NAME? 'RENUMB-A'        Get RENUMB from A device
NAME? N                 Done with RESTART configuration
READY                   Proceed with BASIC language operations
```

* The matrix statements are not available in a system that uses the IOSS package with only 12K of core memory. If the user tries to reply with 2. the question will be repeated but no error message will be given.

The core size entered by the user may be less than or equal to the actual size of the computer memory (12K, 16K, 20K, 24K, 28K, or 32K). By specifying a value less than the true size, the upper postion of memory is made inaccessible to EBASIC. The memory thus saved for other purposes must be a multiple of 1K. There are two primary reasons for saving upper memory:

a. The user may wish to keep BLD II in memory, anticipating that the system will use a language other than EBASIC. He can therefore avoid the inconvenience of using the bootstrap program to load BLD II.

b. The users of the system may wish to write special purpose assembly language subroutines to make use of the reserved memory. For example, it may be used as a common data pool accessible by many subroutines which pass data and/or parameters back and forth (that is, a " blank common" area, familiar to FORTRAN users). This is the fastest way of linking assembly subroutines not included in the same module. That is, the RESTART loader is not a linking loader. (Another way to link subroutines is to pass data and/or parameters through EBASIC CALL statements.)

If the APPEND Library ? query is answered by " Y" , an assembly language subroutine module must be available in the paper tape reader (either Teletype or high-speed reader, as previously selected). Several standard modules are supplied with the system and these are commonly appended to the disc files immediately following a system generation. Modules may also be developed by the user for special purposes. The method of creating assembly language modules is described in the ADAPTS User's Guide.

The standard software modules supplied with each ADAPTS system match the particular system configuration. For example, if the system contains the rack-mounted CRT, the matching " CRT" software module is supplied. A complete list of the standard software modules is given in table 7-10.

During the process of appending relocatable object modules to the library, two possible error messages may be printed on the Teletype.

```
READ ERROR
```

This may be caused by either a bad paper tape or a reader malfunction.

```
SYNTAX ERROR
```

This error is caused by attempting to read an illegally constructed module. See the ADAPTS User's Guide for construction of modules.

In either case, the entire module is ignored and any partial files created on the system file unit are automatically deleted. The question

```
APPEND LIBRARY ?
```

is retyped on the Teletype. This process continues until an N is entered.

All assembly language modules are in relocatable format and may be loaded in any order. They are stored on the system file unit as type S files (see the FLIST command). The name of the subroutine is part of the module itself. It is this name which must be used when the query

```
NAME ?
```

appears on Teletype. The names of some of the Varian supplied subroutines are shown in the preceeding sample dialogue.

Table 7-10. ADAPTS Standard Software Modules

File Name	Description
CRT	CRT Display Subroutines for performing alphagraphic operations on the rack-mounted CRT unit.
KBCRT	CRT Display Subroutines for performing alphagraphic operations on the free-standing keyboard CRT unit.
PLOTTR	Incremental X-Y Plotter Subroutines for alphagraphic operations on the hard-copy plotter.

(continued)

Table 7-10. ADAPTS Standard Software Modules (continued)

File Name	Description
IOSS	Input/Output Subsystem Subroutines for analog and digital input/output operations.
RENUMB	Renumber EBASIC Statements to help edit EBASIC programs.
DEBUGR	Debugger to help troubleshoot relocatable assembly language routines for use with EBASIC.
FFT	Fast Fourier Transform Subroutines, transform and reverse transform, callable from EBASIC.
MAGTAB	Magnetic Tape Unit Filing Subroutines for systems with two magnetic tape units as File A and File B.
CASSAB	Magnetic Cassette Unit Filing Subroutines for systems with two cassette units as File A and File B.
MAGB	Magnetic Tape Unit Filing Subroutines for systems with one magnetic tape unit as File B.
CASSB	Magnetic Cassette Unit Filing Subroutines for systems with one cassette unit as File B.
DISKA	Moving-Head Disc Subroutines for systems with one disc (removable or non-removable) as File A.

Assembly language modules are loaded into computer memory from the disc (system file) or the A or B device. Since modules are appended only to the disc, if they are to be loaded from a cassette (for example), the COPY command must be previously used to transfer the module to the cassette (after a complete system generation).

File Directory Listing (FLIST, FLIST A, FLIST B)

The FLIST command enables the operator to examine the file directories of the system file (FLIST) or the removable files (FLIST A and FLIST B). The directory will be printed by EBASIC in the following format:

```
UNUSED STORAGE = aaaaa

NAME     TYPE     #BLOCKS     #ITEMS
ffffff   t        bbbb        iiii
.....    .        ....        ....
fffff    t        bbbb        iiii
```

where

aaaaa = number of blocks remaining

ffffff = alphanumeric file name

t = file type (P = EBASIC program, S = assembly language subroutine, D = floating point data, I = integer data)

bbbb = number of blocks used by file

iiii = number of values in a standard data file or the (number of characters) / 4 in an EBASIC source program.

If the removable file unit (A or B) is in a 9-track magnetic tape device, the first line of output (UNUSED STORAGE = aaaaa) will not be typed. In this case, the unused storage cannot be determined since reels of magnetic tape may vary in length.

Each program or data file is stored in an integer number of blocks, each of which contains one hundred and seventy-eight (178) 16-bit computer words. The number of items is the actual number of 16-bit words in the file divided by two. Each item represents four characters in an EBASIC program (including all spaces shown in a LIST), one floating point number as stored by a PUT statement, or two data values stored by the IOSS Driver DATIF.

The following example shows an FLIST directory listing.

```
 FLISTB
UNUSED STORAGE=3056
NAME      TYPE      #BLOCKS      #ITEMS
ANUITY    P         8              670
BELOOP    P         12            1026
AMAZIN    P         20            1729
JAKCUS    P         14            1193
CLNDAR    P         11             910
DECIDE    P         10             863
FSMMIN    P         15            1334
GRADFR    P         6              483
HISTO     P         8              688
HOOK      P         12            1064
TRUINT    P         7              543
```

Initialization of Removable File Media (CLEAR A, CLEAR B)

The CLEAR command initializes the selected removable storage medium (A or B) so that it may be used by the system. When using a cassette cartridge or a magnetic tape for the first time, the user must type either

```
CLEAR A or CLEAR B
```

before any programs or data can be stored on the medium. When the CLEAR command is used, the following warning message will be printed by EBASIC:

```
FILES ON MEDIA WILL BE DESTROYED
```

If the user wishes to go ahead with the CLEAR operation, he depresses the carriage RETURN key; if he wishes to abort the CLEAR command, he hits the ESC key. If the medium has been previously unused, the message will not have meaning. However, as indicated, the CLEAR command may also be used to delete all old files on the medium and make medium available for re-use.

A CLEAR command not followed by either A or B will not be recognized. Thus, CLEAR can never be used to destroy the system files. Because imbedded spaces are ignored in the command it may be typed as CLEARA or CLEARB.

Storage and Recovery of Program Files

The SAVE and LOAD commands are used to transfer program files (only) back and forth between computer memory and any of the bulk storage units (system file, unit A, or unit B). Data files are handled by PUT and GET commands.

SAVE

The SAVE command has three forms:

SAVE " ffffff"	Save on system file
SAVE " ffffff-A"	Save on unit A
SAVE " ffffff-B"	Save on unit B

The file name (ffffff) may have from one to six characters, the first of which must be a letter (A to Z). The last five characters (optional) may be letters or digits (0 to 9). The file name is separated from the unit identifier by a dash. The entire identifier is enclosed in quotation marks. Blank spaces are not allowed within the quotation marks.

If the file name specified is identical to one which is already present on the designated bulk storage device, the following message will be printed:

```
OLD FILE ?
```

This gives the user an opportunity to abort the SAVE command, which he may do by striking the ESC key. If he wishes to replace the old file, he may do so by using carriage RETURN. When the program has been " saved" , the Teletype bell will ring once.

A program may be saved under the same name on each of the three bulk storage devices in the system. Only EBASIC programs can be saved; assembly language subroutines, for example, may not be stored on any of the bulk storage devices with the SAVE command.

LOAD

The LOAD command has three forms

LOAD " ffffff"	Loads program from system file
LOAD " ffffff-A"	Loads program from unit A
LOAD " ffffff-B"	Loads program from unit B

A data file cannot be loaded with the LOAD command. Data files are recovered through the GET command.

The file name (ffffff) followed by the unit identifier A or B (removable media only) must be enclosed in quotation marks. Spaces are not permitted between quotation marks. Only program files listed in the directory may be loaded.

When programs are LOADed, each line is brought in the file unit in the same manner as lines are accepted from the Teletype keyboard. If a statement in the program file being loaded has the same line number as the program in the computer, it will overlay the old statement. Otherwise, it will insert the new statement in the program. As each line is brought in it is checked for construction and rejected if in error. For example, if a program being loaded has a CALL statement to a subroutine not loaded during the restart phase, then that statement will not be loaded. An error message will be printed and the load process will terminate. Control returns to the READY state.

Since a program being loaded may mix statements inconveniently with the program already in the computer, it is customary to precede the LOAD command with a SCRATCH.

The LOAD command may be used as a program statement to perform overlays. This " dynamic use" of LOAD is explained in a later section.

Creation and Use of Data Files

Five commands are available to handle data files:

ASSIGN	Associates one, two, or three file names with corresponding file numbers (1, 2, or 3)
OPEN	Initializes a data file for read or write access

(continued)

PUT	Places a list of values into a numbered data file (1, 2 or 3); sequential access
GET	Assigns sequential values from a numbered data file (1, 2 or 3) to a variable list
CLOSE	Terminates input or output to a specified data file (1, 2 or 3)

Data is transferred to and from the bulk storage devices through file buffers, which are sections of computer memory assigned for that purpose. Three file buffers are implemented and they are referenced by a number (1, 2, or 3) in the program. The numbers are assigned alphanumeric names just prior to the running of the program. In this way a more general program can be written. For example, the program might use file 1 for input, file 2 for output, and file 3 for a scratch file.

The following example is presented without comment. The use of ASSIGN, OPEN, CLOSE, PUT and GET will become clear in the next four sections.

```
 LIST
10    OPEN  1, 0
20    FOR  I= 1 TO 7
30    PUT  1, I, I, * I
40    NEXT I
50    CLOSE  1,R
60    PRINT 'ITEMS ON CLOSE=';R
70    OPEN  1, 1,N
80    PRINT 'ITEMS ON OPEN=';N
85    PRINT 'X', 'X↑2'
90    FOR I= 1 TO N/ 2
100   GET 1, X,Y
110   PRINT X,Y
120   NEXT I
130   CLOSE 1
140   END
 ASSIGN  'DATA-A'=1
1=NEW
 RUN
ITEMS ON CLOSE= 14
ITEMS ON OPEN= 14
X                 X↑2
 1                 1
 2                 4
 3                 9
 4                 16
 5                 25
 6                 36
 7                 49
```

(continued)

```
READY
 FLISTA
UNUSED STORAGE =592
NAME       Type       #BLOCKS       #ITEMS
ANOVER       P          29            2561
DATA         D          1              14
 ASSIGN 'DATA-A' =1
1=OLD
```

ASSIGN

The ASSIGN command assigns or associates alphanumeric file names with the file numbers 1, 2, and 3. Examples:

```
ASSIGN 'DATA' = 1
ASSIGN 'DATA-A' = 2, 'JUNK-B' = 3
ASSIGN 'SCRAP' = 1, 'TEST' = 2, 'MICE-A' = 3
```

The command is normally given just prior to running the program. The OPEN, CLOSE, PUT, and GET statements make use of the logical connection between the file numbers and the file names. The ASSIGN statement may be made a numbered statement in the program but this usage destroys the generality of the scheme. For example, a program may use files 1 and 2 for data, and 3 for scratch. By deferring the naming of the files until execution time, many data files may be used by one program.

The ASSIGN command is used for data files only; programs are never assigned file numbers. Up to three files may be assigned on a single line and, in fact, file numbers not referenced in an ASSIGN statement are assumed to be not assigned.

For example, if the following statement is typed,

```
ASSIGN 'ONE' = 1
```

and then

```
ASSIGN 'TWO' = 2
```

the effect of the second assignment is to assign the name TWO to file 2 and to cancel the previous assignment of name ONE to file 1. If we wish to assign names to both 1 and 2 then we must do it in a single statement, for example,

```
ASSIGN 'ONE' = 1, 'TWO' = 2
```

When the ASSIGN command is given, the file directory is searched to determine if the names have been previously used. Suppose that in the last example ONE had been used before but TWO had not been used, then the computer would print the the following message:

```
1 = OLD, 2 = NEW
```

Of course, if the ASSIGN command is given a line number in the program, the message is only an indication that the program is operating. But if the ASSIGN command is given prior to run, it gives the user a second chance to decide on a new assignment, since no action will be taken until he types RUN. An " old" file name is perfectly acceptable, but it means that the file may be altered by running the program.

OPEN

The OPEN statement is used to initialize a file number for either write or read access. After an OPEN statement the GET or PUT pointer is set to the first item in the file.

To PUT items into a file (write access) we must use a statement of the form:

OPEN file number, 0

To GET items from a file (read access) we must use a statement of the form:

OPEN file number, 1, *optional variable*

The file number may be any variable or formula which when evaluated and truncated to an integer, is equal to 1, 2, or 3. When a file is opened for read access, the optional variable is set equal to the current number of items in the file.

Examples:

```
10 REM OPEN FILE 1 FOR GET
20 OPEN 1, 1, N
30 REM N IS NOW EQUAL TO NO. ITEMS IN FILE 1
40 REM OPEN FILE 2 FOR PUT
50 OPEN 2, 0
```

Each time a file is OPENed for either GET or PUT operations, the data pointer is set to the first item in the file.

Only one file at a time may be OPENed on a magnetic tape unit. If a file is OPENed on a magnetic tape unit, the following statements cannot be issued specifying a file on the same unit: COPY, LOAD, SAVE, or DELETE. Also, the FLIST command (FLISTA or FLISTB) will not display the contents of that bulk storage unit.

A given file may be OPENed for read access on more than one file number. A file OPENed for write access, however, may be ASSIGNed to only one file number at a time.

PUT

The PUT statement is used to place values in a data file previously opened for write access. Its form is

PUT file number, list

where the file number may be any variable or formula which, when truncated to an integer value, is equal to 1, 2, or 3. The list of items, which are placed sequentially in the file, may also be variables or formulae. For example:

```
10 REM OPEN FILE 2 FOR WRITE
20 OPEN 2,0
30 FOR I = 1 TO 10
40 PUT2, I, I + 2, SQR(I)
50 NEXT I
60 CLOSE 2,N
```

This example places 30 values into file 2. The values are PUT sequentially in the order in which they appear in the list following the file number. Here 1, 1, and 1 are the first items in the file and 2, 4, and $\sqrt{2}$ are the fourth, fifth, and sixth items. The OPEN statement initializes the pointer to the first item in the file and each value PUT into the file increments the pointer by one.

GET

The GET statement is used to recover items from a data file previously opened for read access. Its form is

GET file number, list

where the file number may be any variable or formula which, when truncated to an integer value, is equal to 1, 2, or 3. The list contains one or more variables which receive the items " gotten" from the file in sequential manner. For example:

```
10 REM OPEN FILE 3 FOR READ
20 OPEN 3, 1, N
30 GET 3, X, Y, Z
```

The first three items in the file are assigned to X, Y, and Z. The data pointer is initialized to the first item in the file by the OPEN statement. Each time an item is obtained from the file the pointer is incremented by one. **Note: Items retrieved from an integer data file (type I) are converted to floating point automatically by the GET.**

CLOSE

The CLOSE statement terminates GET or PUT operations on a file. Its form is

CLOSE file number, *optional variable*

where the file number may be any variable or formula which when truncated to an integer evaluates to 1, 2, or 3. The optional variable is set equal to the number of items in the file

at time of CLOSE; the optional variable may be used only when terminating access to a file OPENed for write.

EXAMPLE:

```
300 CLOSE 1, N
```

An implicit CLOSE is performed on every open file when a program is aborted through use of the ESC key or through a program error. CLOSE is also performed on all open files when a normal run is completed.

COPY Statement

The COPY statement permits the user to transfer any file from one bulk storage device to another bulk storage device. It is also possible to copy the file on the same bulk storage device unless that device is a magnetic tape unit.

The form of the COPY statement is

COPY file name TO file name

where the first file name is the source and the second file is the sink. If the sink file name is found to already exist in the file directory of the designated unit, the message

```
OLD FILE?
```

is printed. This gives the user a chance to abort the COPY command by striking the ESC key. Otherwise, the user may give the go-ahead signal by striking carriage RETURN.

Examples:

```
COPY 'DATA' TO 'DATA-A'
COPY 'PROG1' TO 'PROGZ'
COPY 'SAMP-A' TO 'SAMP-B'
COPY 'TEST' TO 'TEST'
```

Notice that all file names are constructed according to the rules specified under SAVE.

The last example is legal but of no use since it will result in EBASIC reading the file into a core memory buffer and writing it out again on the disc in the same place. Only one file named TEST will appear in the file directory.

The source file may be on any of the bulk storage devices: system file, unit A, or unit B. The sink file may also be any of the bulk storage devices except that a source file on magnetic tape may not be copied onto the magnetic tape in a single operation; it must

first be transferred to another unit (for example, the system file) and then may be transfered back to the tape.

Note: The COPY statement may be used as a numbered statement in the program if and only if the files have been previously CLOSEd. Otherwise an error will be reported.

DELETE Statement

The DELETE statement may be used to erase a specified program file, subroutine file, or data file from any of the bulk storage devices on the system. For example:

```
DELETE 'SAMPLE'
```

The file name must be enclosed in quotation marks and is constructed according to the rules given under the SAVE command. Only one file may be deleted at a time. If the name specified is on the file unit, then the system will respond to the DELETE statement by printing

```
OLD FILE?
```

If the user wishes to go ahead with the command, he depresses carriage RETURN; if he wishes to abort the command, he uses the ESC key.

When a file is deleted, the space previously used by it is released for re-use, unless the file was on a reel of magnetic tape.

Program Overlays--Dynamic Use of LOAD

The LOAD command may be used as a numbered statement in the program. This facility permits the programmer to write EBASIC programs in segments which may be SAVEd on bulk storage units and LOADed into the computer only as they are needed. A program written in this manner usually includes one or more REMOVE commands to release computer memory for the incoming segment and to avoid an " inconvenient" mixing of current statements with incoming statements. Statements in the current segment which are not overlaid by statements in the incoming segment or REMOVEd before the LOAD is executed will remain after the LOAD has been completed.

The conditions and restrictions which govern the use of the LOAD statement in a program are as follows:

a. A LOAD statement should not be written inside a FOR-NEXT loop or a subroutine.

b. Simple variables and array variables retain their values from one segment to the next, unless changed by the program.

(continued)

c. A DIM statement must be overlaid by the incoming segment or REMOVEd prior to the LOAD statement.

d. A user-defined function (DEF statement) remains defined from segment to segment only if it is not overlaid or REMOVEd.

e. An implicit RESTORE is performed when a LOAD statement is executed. This initializes the pointer to the first number in the data pool after the LOAD has been completed.

After a successful LOAD, control will pass to the next numbered statement in the program.

The procedure for incorporating LOAD statement into the program is illustrated in the following example. In this example, the segments of the program are called " pages" for convenient reference. Note that statement 40 in PAGE01 had to be either REMOVEd (as was done) or overlaid by PAGE02 or else error message 45 would have been reported.

```
 SCRATCH

READY
 LOAD 'PAGE02'
 LIST
1  REM PAGE02
190   DIM B (5)
200   READ B ( 1),B( 2),B( 3),B( 4),B( 5)
210   PRINT 'A(I)', 'B(I)','FNX(A(I))','FNX(B(I))
220   FOR I= 1 TO 5
230   PRINT A(I),B(I),FNS(A(I)),FNX(B(I))
240   NEXT I
400   END
 SCRATCH

READY
 LOAD 'PAGE01'
 LIST
1  REM PAGE01
10   DEFFN X(X)=X ↑ 2
20   DATA 1, 2, 3, 4, 5
30   READ A( 1),A( 2),A( 3),A( 4),A( 5)
40   DIM A(5)
50   REMOVE 25 TO 50
60   LOAD 'PAGE02'
70   REMOVE 60 TO 70
80   REM WILL NOT BE OVERLAID TO ILLUSTRATE PROCESS.
400   END
```

(continued)

```
 RUN
A(I)        B(I)        FNX(A(I))        FNX(B(I))
 1           1           1                1
 2           2           3.99999          3.99999
 3           3           8.99997          8.99997
 4           4           15.9999          15.9999
 5           5           24.9999          24.9999
READY
 LIST
1  REM PAGE02
10  DEFFN X(X)=X ↑ 2
20  DATA  1, 2, 3, 4, 5
80  REM WILL NOT BE OVERLAID TO ILLUSTRATE PROCESS.
190  DIM B(5)
200  READ B( 1),B( 2),B( 3),B( 4),B( 5)
210  PRINT 'A(I)','B(I)','FNX(A(I))','FNX(B(I))'
220  FOR I= 1 TO 5
230  PRINT A(I),B(I),FNX(AI)),FNX(B(I))
240  NEXT I
400  END
```

PROGRAMMING THE INTERFACE CONSOLE

The Interface Console (IFC) is programmed via CALL statements to the Input/Output SubSystem (IOSS) drivers, which are assembly language subroutines. The IOSS drivers and their functions are:

DATAI	Inputs data to an array
DATIF	Inputs data to a file
DATAO	Outputs data from an array
PULSE	Operates a control line output
STATUS	Senses a status line input

Analog and Digital Channels

The IOSS drivers (assembly language subroutines) DATAI, DATIF, and DATAO allow input or output over one or more channels in a single CALL statement. The channel may be handling analog or digital data as determined by the factory wiring. Each channel is assigned a number, to which the subroutines may refer, but only the programmer knows whether the channel number is assigned to analog data or digital data.

The analog inputs are multiplexed to a 13-bit (binary) analog-to-digital converter, which accepts signals of ± 10 volts full scale. Thus the full scale values (numbers) which are read on these channels are +4095 (+10 volts) and −4096 (−10 volts). The negative limit is slightly larger than the positive limit because a two's complement format is used to represent raw input data. Data is then transformed to floating point format so that it may be used by the EBASIC program.

The analog outputs are multiplexed to 12-bit (binary) digital-to analog converters. Again, two's complement format is used by the converter circuitry so that full scale is +2047 (+10 volts) and −2048 (−10 volts). If a positive value greater than 2047 is output or a negative value less −2048 is output, then the higher bits are ignored. This causes 2048 to be equivalent to +1 and −2049 to be equivalent to −1 (for example). A number N outside the range

$$-32768 \leq N \leq 32767$$

cannot be contained in the 16-bit computer word used. DATAO will report the attempt to output numbers outside this range as an error.

The digital inputs or outputs are 16-bits in binary two's complement format. Again, the maximum values which can be contained in 16 parallel bits are +32767 and −32768. In two's complement format, negative numbers are represented as the one's complement form plus one. A negative number in one's complement form is obtained by taking the number in binary form and replacing every "1" with a "0" and vice versa.

Thus we have, for example,

```
+1 = 0 000 000 000 000 001
-1 = 1 111 111 111 111 110          (one's complement)
-1 = 1 111 111 111 111 111          (two's complement)
```

The CALL statements for DATAI, DATIF, and DATAO are very similar. DATAI may be used as an example:

```
10 CALL DATAI, M(1), A(1), N, S, C
```

where

M(1) = first element of a channel selector array
A(1) = first element of the data storage array
N = number of data points
S = scan time, which is the time (in microseconds) to scan all channels in the channel selector array
C = capture delay, which is the time delay (in microseconds) between channels in the scan

Notice that the CALL statements for DATAO and DATIF are very similar, for example,

```
20 CALL DATAO, M(1), A(1), N, S, C
30 CALL DATIF, M(1), 1, N, S, C
```

where all the above definitions of the CALL arguments apply except that a file number has replaced A(1) in the CALL to DATIF. For DATAO, A(1) is the first element of an output array instead of an input array as used with DATAI.

The simple variables and array variables used above are illustrations only. Any variable names may be used except that the first two arguments must be array variables. Furthermore, the dimensions of these arrays must have been previously declared in a DIM statement.

Note: When using DATAI or DATIF, the programmer must consider the SCAN SYNC line on the IFC. For a description of its use, see the section on scan time.

Channel Selector Array

The first argument in the CALL statement to DATAI, DATAO, or DATIF is an array variable which is the first element in a channel selector array, for example,

```
10 CALL DATAI, M(1), A(1), N, S, C
```

For convenience, this element will be referred to as M(1) and array M is then the channel selector array. The channel selector array specifies the channel numbers over which data is to be transferred and whether the specified channels are to be accessed in every scan, every other scan, every third scan, etc.

In setting up the channel selector array the user must decide whether to use sequential addressing of channels or random addressing. Sequential scans are easier to set up but require data rates to be the same on every channel specified. Channels may not be skipped in a sequential scan; the scan occurs from channel 1 to the last channel given. Random addressing allows different data rates on various channels and permits use of only those channels absolutely specified.

The channels are, for the most part, the numbers given on the IFC. For DATAI channels 1-16 are assigned to analog inputs 1-16. The digital input will be channel 17, unless more than sixteen analog inputs are installed on the system. Of course, more than sixteen analog inputs are not available on a single IFC. But if the analog channels are expanded then the digital channels will be numbered following the highest numbered analog channel. A similar situation exists for the output channels. Output channels 1-8 are assigned to analog outputs (as marked on the IFC) and the digital output channel is assigned number 9, unless the analog outputs are expanded beyond 8.

SEQUENTIAL SCAN

The sequential scan of input or output channels is set up in the following manner:

M(1) = 1
M(2) = last channel in scan

When this type of channel selector array is specified, all channels from channel 1 to the last channel will be accessed in every scan time. The scan time is given as the fourth argument in the CALL statement.

The programmer must take the number of channels to be scanned into account when assigning values to the scan time and capture delay.

RANDOM SCAN

The random scan of input or output channels is set in the following manner:

```
M(1) = number of items to follow
M(2) = -1 (access every scan)
...
...          channels to be accessed in every scan; channel numbers
             in ascending order
...
M(1) = -2 (access every second scan)
...
...          channels to be accessed every second scan; channel numbers
             in ascending order
...
M(J) = -3 (access every third scan)
...
...          channels to be accessed every third scan; channel numbers
             in ascending order
...
etc.
```

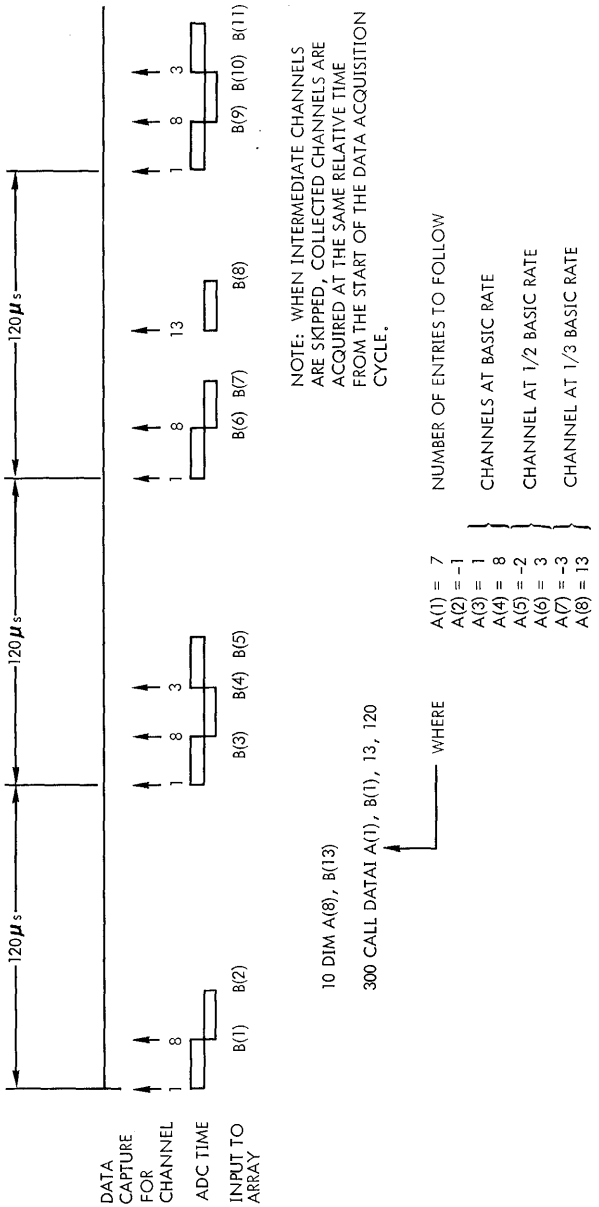
Note that a negative integer is used to specify the random scan. For a random scan, M(2) is a -1 and succeeding positive entries up to the next negative number are accessed on every scan. Similarly, succeeding negative entries specify how often other channels are to be accessed.

Example:

```
M(1) = 5
M(2) = -1
M(3) = 1
M(4) = 3
M(5) = -2
M(6) = 2
```

In this example channels 1 and 3 will be accessed every scan and channel 2 will be accessed on the even scans. A detailed example of a random scan is illustrated in figure 7-1.

The specifications for the channel selector will affect the specifications for scan time and capture delay, as described in the appropriate sections of this manual.



VTII-1917

Figure 7-1. Random Scan Mode Example

Data Storage Array (DATAI and DATAO)

The data storage array is identified by the second argument in the CALL statement. This array variable, which will be referred to as A(1), is the first location to be used for input (DATAI) or the first location to be used for output (DATAO). There is no reason why the first element cannot be A(2) or B(10) or C(1,20). A double-subscripted array variable must be used if the program is to output more than 255 values in a single CALL statement, however. In any case the array dimensions must be declared in the program by a DIM statement.

If using a double-subscripted array variable, the programmer should be aware that input or output is by columns, not by rows. For example, an array A with 3 rows and 2 columns (DIM A (3,2)) contains array elements which are used in the following order:

A (1,1)
A (2,1)
A (3,1)
A (1,2)
A (2,2)
A (3,2)

File Number (DATIF)

When using DATIF, the second argument in the CALL statement may be a constant, a variable or a fomula which has a truncated value equal to one of the three file numbers (1, 2 or 3).

OPEN and CLOSE statements for the file used should not be given when DATIF is called. DATIF performs its own OPEN and CLOSE operations. However, an alphanumeric file name must be assigned to the file number via the ASSIGN command. This is commonly done prior to running the program. If the file to be used is OPEN for read or write when a CALL for DATIF is encountered, the file is deleted and reOPENed in the write mode.

The DATIF command can be used with all bulk file units except cassettes.

Number of Data Points

The third argument in the CALL statement to DATAI, DATAO, or DATIF is the number of data points to be transferred. A variable, formula, or constant may be used; the value is truncated to an integer. The number of data points, N, may be as large as 32767 for DATIF; for DATAI and DATAO, N is limited to the available computer memory.

The number of data points specified must not exceed the number of elements in the array which are given in a DIM statement. If the number of data points exceeds the number of

elements in the array, there are three possible destructive consequences when using DATAI:

a. The input may overflow into the next array(s).

b. The input may overflow all the arrays and destroy the EBASIC program, thus putting the computer into step mode. This will necessitate the manual restart of the computer.

c. The input may overflow into the assembly language subroutines and possibly into memory locations used by EBASIC itself. In either case the restart bootstrap will have to be used to recover.

Scan Time

The fourth argument in the CALL statement specifies the number of microseconds in a data cycle. For convenience, this argument will be referred to as S. The scan time is truncated to an integer, which must be in the range

$$32767 \geq S \geq T * N$$

where N = number of channels in the data cycle
T = minimum time to transfer a data point over one channel

Refer to figure 7-2 for a detailed illustration of scan time and capture delay.

The scan time may also be set equal to zero, which permits the external data source to give a " data present" signal for every single data point to be input. SCAN SYNC is raised to +5 volts each time the computer is ready to accept another data point. SCAN SYNC is externally grounded to synchronize each scan. SCAN SYNC is also used when the scan time is not zero except that this line need be grounded only once to cause the computer to input data to the entire array at the rate indicated. SCAN SYNC is used only with DATAI and DATIF (not DATAO).

The number of channels, N, in the data cycle is determined from the channel selector array. When using sequential scan, the number of channels is simply the last channel number specified. When using random channel selection, the number of channels is the maximum number of channels which will be used in any one scan time.

The minimum time (T) required to transfer data over a channel depends on the subroutine used and storage media. A summary of values assigned to T is given in table 7-11.

If the value given for the scan time falls within the bounds given earlier, then all channels specified in the channel selector array may be accessed in the time allotted for the scan. But if the scan time is not sufficient, then the computer will take as much time as is required and no error message is returned. The following example shows how a value for the scan time is computed.

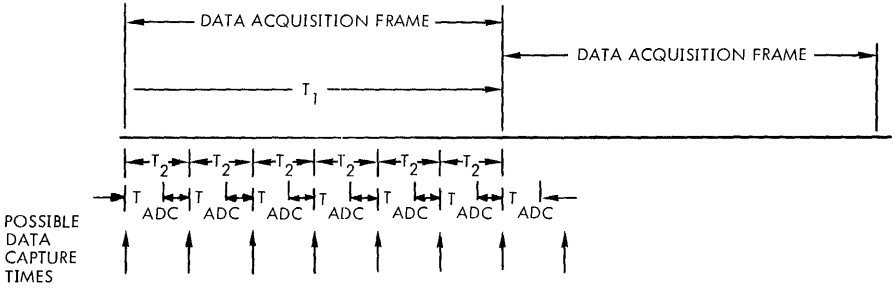
The channel selector array is

M(1) = 5
M(2) = -1
M(3) = 1
M(4) = 3
M(5) = -2
M(6) = 7

DATIF is used to input to a magnetic tape unit. (25 ips)

S min = N * T
where T = 333
N = 3
therefore, S min = 3*333 = 1000 microseconds

A large value for scan time may be specified, if desired.



MEANING OF SYMBOLS

T_1	DATA ACQUISITION CYCLE TIME INTERVAL (SCAN TIME)
T_2	TIME INTERVAL BETWEEN EACH DATA INPUT OPERATION (CAPTURE DELAY)
T_{ADC}	TIME TO COMPLETE ONE ANALOG-TO-DIGITAL CONVERSION
N	MAXIMUM NUMBER OF CHANNELS TO BE COLLECTED IN ONE T_1 TIME

IMPORTANT RELATIONSHIPS

IF T_2 NOT SUPPLIED (EQUIVALENT TO SETTING $T_2 = T_{ADC}$)

$T_2 \geq N*T_{ADC}$

IF T_2 SUPPLIED

$T_1/T_2 \geq N$ (MUST BE INTEGER RESULT)

$T_2 \geq T_{ADC}$

VTII-1918

Figure 7-2. Time Specification Arguments

Table 7-11. Minimum Times to Transfer Data Over a Channel for 620/L-100 or 620/f-100.

IOSS Driver	Input/Output Data Storage	Minimum Transfer Time (T) in usec	Equivalent Throughput in kHz
DATAI	Core memory array	50	20
DATAO	Core memory array	150	15
DATIF	Moving-head disc	667	1.5
	Fixed-head disc	225	4.5
	25 ips tape deck (9 track)	333	3.0
	37.5 ips tape deck (9 track)	225	4.5

Capture Delay

The fifth argument in the CALL statement specifies the time in microseconds between adjacent channels in the scan. The capture delay must be an integer sub-multiple of the scan time.

The capture delay is optional for DATAI and DATAO. If it is not specified then data will be input or output as fast as possible. However, when using DATAI and selecting more than one channel, the failure to specify the capture delay will result in data points unevenly spaced in time. This is because the minimum time to switch channels is a variable (if not controlled) although it will always be less than the value given in table 7-11. However, when using DATAO, even though capture delay is not specified, the data points will always be equally spaced in time.

To continue the example chosen in the section on scan time, the channel selector array is

M(1) = 5
M(2) = -1
M(3) = 1
M(4) = 3
M(5) = -2
M(6) = 7

DATIF will be used for input to a tape unit (25 ips). The minimum scan time we can choose is (from table 7-11) 1000 microseconds. For illustration 2000 is chosen, which gives a throughput of 500 Hz for each channel included in every scan. For capture delay some value is required which when multiplied by an integer is equal to 2000. Our choices are somewhat limited:

2000/6 = 333.3	(Not an integer)
2000/5 = 400	(O.K. Minimum time to transfer)
2000/4 = 500	(O.K.)
2000/3 = 666.7	(Not an integer)
2000/2 = 1000	(Cannot fit 3 channels in scan)
2000/1 = 2000	(Cannot fit 3 channels in scan).

Thus, the only valid choices for the capture delay with 3 channels and a scan time of 2000 are 400 and 500. If, however, input to the system file is performed, we could choose a capture delay of 250 (250 x 8 = 2000). The graphic representation of this solution is shown in figure 7-3.

Control and Status Line Operation (PULSE, STATUS)

The IOSS drivers (assembly language subroutines) PULSE and STATUS allow the programmer to operate the digital output CONTROL lines and the digital input STATUS lines. Each deals with a single bit of information (0 or 1), whereas DATAI, DATAO, and DATIF deal with many bits in parallel (up to 16).

The statement

```
10 CALL PULSE, N
```

turns on the NPN transistor switch (to ground) which is control line N. Each transistor switch is reset by an external input to ground on the appropriate connector.

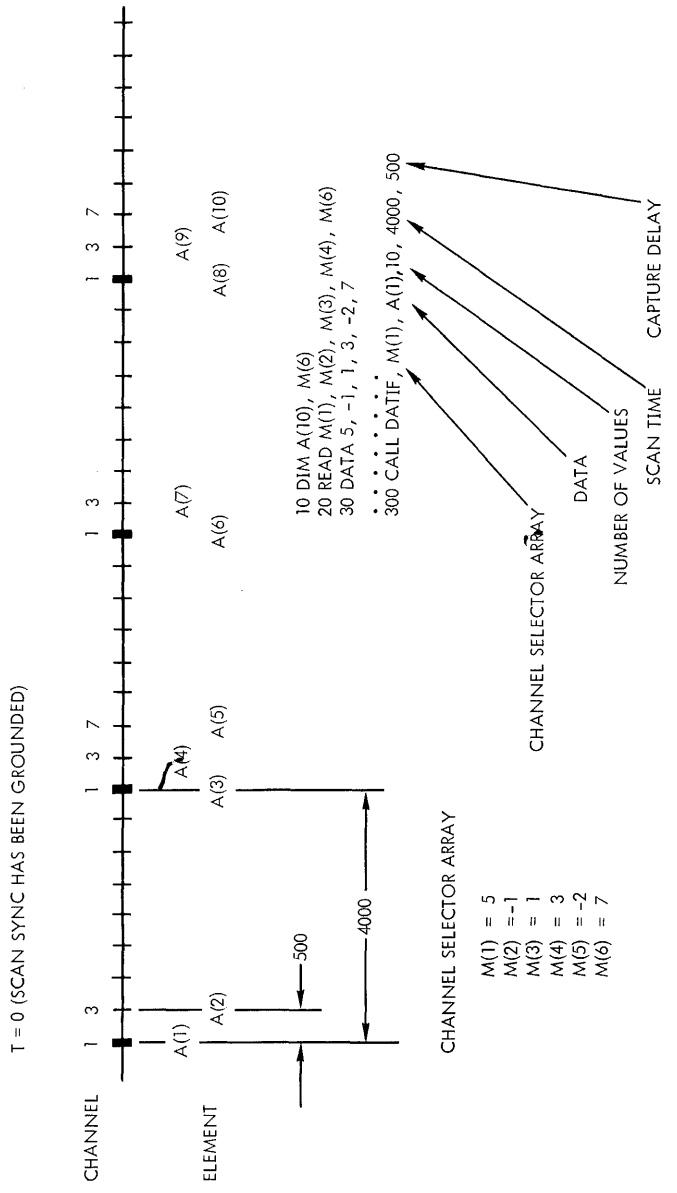
Control lines 1 through 8 are available on the IFC. Lines 9 through 56 are available as options but they must be accessed on the printed circuit boards.

The STATUS inputs permit the user to examine a single bit at a time by calling the IOSS Driver STATUS. For example

```
100 CALL STATUS, 2, N
```

This CALL examines the input to status input number 2 and sets the variable N equal to 1 if the input is at ground (true). If the input is high (false), the variable N will be set equal to 0 (zero). The program may then make a decision based on this value.

The status inputs may also be used in an interrupt mode. However, the user must generate his own assembly language subroutines for this purpose (see the ADAPTS User's Guide).



VTII-1919

Figure 7-3. Timing Diagram for a Sample CALL Statement Using DATIF

INFORMATION DISPLAY ON OSCILLOSCOPE

A summary of the commands for operating the oscilloscope is given in table 7-12. The oscilloscope controls are operated by assembly language subroutines. The oscilloscope, Model A-620/73, is modified to match the signal levels at the hardware controller. The controller is similar to an analog output module and, therefore, the PULSE and STATUS subroutines may be used to operate some of the features of the A-620/73 such as the STORE/NON-STORE choice of modes. These will be explained later.

Both alphanumeric characters and graphical figures (dots and lines) may be output to the oscilloscope. Characters are generated by a software character generator. The size of the characters is independently controlled by the SIZE subroutine.

All alphanumeric output, whether typed from the keyboard or printed with a PRINT statement, is directed either to the Teletype or to the oscilloscope by the routines TTY and CRT. Alphanumerics cannot be printed simultaneously on the screen and on the Teletype page.

The screen is initialized for a fresh page of alphanumeric output by CTRL L (which may be included in a PRINT statement between quotation marks). If outputting alphanumeric information only, each line will be automatically positioned on the page. Character size 2 permits 72 columns on the screen - the same as a Teletype page. When any attempt is made to print below the page, the software character generator rings the Teletype bell and waits for a go-ahead signal from the operator. The go-head signal, which is any key on the Teletype, causes the screen to be erased, the beam to be re-positioned to the upper left corner, and the information display to be resumed.

The POS subroutine may be used to position the beam before printing characters or drawing a straight line. In order to use ORG, POS, POINT, and VECT the user needs to understand the coordinate system of the screen. With the origin set to (0,0), the screen is organized so that

$$-512 \leq X \leq 511$$
$$-400 \leq Y \leq 399$$

where X has a larger range because the screen is rectangular. Equal increments in X and Y cause the beam to be moved at a 45 degree angle.

The actual X and Y coordinates may be used and thus the values are modulo 511 (- 512 for negative values) up to the limit which can be contained in a 16-bit word length; the high order bits (except sign) are simply not used by the converter. Thus, for example, X = 512 is equivalent to X = 0, and X = 513 is equivalent to X = 1. This modulus feature could be used to expand a graphic display by using variables as outer bounds, which might be assigned values as large as ± 32767. However, this technique might cause the routine to " waste" time by writing off-screen.

Table 7-12. EBASIC Oscilloscope Commands

Command	Function
CALL CRT	Switches alphanumeric output stream to the CRT
CALL TTY	Switches alphanumeric output stream to the Teletype
CALL INIT	Initializes the CRT by erasing the screen; positioning the beam to (0,0); and resetting the origin to (0,0), the character size to (2), the scale factor to (100), and the zoom factor to (100).
CALL ERASE	Erases the screen, leaves the current X, Y position unchanged, and places the CRT in the ready-to-write mode
CALL SIZE,S	Sets the size of alphanumeric characters according to (S) which may be a formula, but must evaluate to a positive integer between 1 and 80. Once set, SIZE remains unchanged until altered by another SIZE or ZOOM command. Initially, S = 2 which allows 44 lines of 72 characters each to be written on the screen. The ZOOM command effects SIZE as follows: $\text{new size} = \text{greatest integer}\left(\frac{(S * \text{ZOOM factor})}{(100)}\right)$
CALL SCALE,SF	Sets the scale factor (SF) to enlarge or reduce graphic output only while alphanumerics simply change absolute position. Once set, SCALE remains unchanged until altered by another SCALE or ZOOM command. Initially, SF = 100; thus 50 reduces the graphic output by a factor of 2 and 200 enlarges it by a factor of 2. The ZOOM command affects SCALE as follows: $\text{new scale} = \text{greatest integer}\left(\frac{(SF * \text{ZOOM factor})}{(100)}\right)$ SF can be a formula, but must evaluate within the range −32768 to 32767
CALL ZOOM,ZF	Sets the zoom factor (ZF) to enlarge or reduce the entire alpha-graphic presentation. Once set, ZOOM remains unchanged until altered by another ZOOM command. Initially, ZF = 100; thus if ZF = 50 and SF = 300, the new scale will be 150

(continued)

Table 7-12. EBASIC Oscilloscope Commands *(continued)*

Command	Function
	and the alpha-graphic output will be enlarged by a factor of 1.5. (ZF) can be a formula, but must evaluate within the range −32768 to 32767.
CALL ORG,X,Y,M	Stores bias values for X and Y coordinates permitting translation of the origin of the coordinate system to any desired point. Once set, the bias values remain unchanged until altered by another ORG command. Initial bias values are both zero. (X) and (Y) can be formulas, but must evaluate within the range −32768 to 32767. (M) must evaluate to zero (specifying absolute) or one (relative)
CALL POS,X,Y,M CALL POINT, X Y,M	POS positions the beam; POINT positions the beam and writes a point on the screen. Positioning to the new X- and Y- coordinates can be absolute (if M = 0) or relative (if M = 1)
CALL VECT, X, Y,M	Draws a straight line from the present beam position to the new X- and Y- coordinates absolute (if M = 0) or relative (if M = 1)
RETURN key	Pressing the RETURN key on the Teletype keyboard (anytime after the CALL CRT command) positions the beam to: X = −512 Y = −unchanged
CTRL,FORM key	Pressing and holding the CTRL key, then pressing the FORM key erases the CRT and positions the beam to: X = −512 Y = 399
LINE FEED key	Pressing the LINE FEED key positions the beam to: X = unchanged Y = current Y - 10 * S if the new Y < −400 the Teletype bell rings and pressing any key executes a CTRL,FORM

(continued)

Table 7-12. EBASIC Oscilloscope Commands *(continued)*

Command	Function
Auxillary Commands for Oscilloscope	
CALL PULSE,59	Pulse the Z-axis, i.e., write a dot
CALL PULSE,GO	Erase the screen. Does not change the beam position; does not check for read-to-write condition
CALL STATUS, 57, (TF)	Sense the erase interval. If complete, set (TF) = 1; else set (TF) = 0
CALL PULSE, 61	Select non-store mode
CALL PULSE, 62	Select store mode
CALL PULSE, 63	Select write-thru mode. Effective only if store mode
CALL PULSE, 64	Select non-write-thru mode

Alphanumeric characters are printed in a 5 x 7 dot matrix which is expanded by the argument given in the CALL to the subroutine SIZE.

In order to lengthen the life of the oscilloscope tube, the display unit has a view/hold feature which causes the stored display to decrease in intensity approximately 60 seconds after either of the following:

a. the VIEW button on the front panel has been pressed; or

b. output to the screen has ceased.

The unit may be returned to the view mode by pressing the VIEW button or by writing any information on or off screen. If the user wishes to override the normal return to hold mode after 60 seconds, then he may position the beam near the edge of the screen and repetitively operate line 59 with the command

```
CALL PULSE, 59
```

Several other auxiliary commands may be useful in programming. The NON-STORE and WRITE-THRU modes are of particular interest. While in NON-STORE mode, the oscilloscope screen will not store information; however, previous information written in STORE mode is retained, although not visible. Therefore, the user may write on the screen in STORE mode and then switch back and forth between NON-STORE and STORE modes. This will cause the stored display to blink on and off in an attention-getting manner.

The WRITE-THRU mode may be used while the unit is in STORE mode to display the beam position without storing the information on the screen. The following example shows how to position the beam at the center of the screen and display an " x" without storing it on the screen. The " x" will be very fuzzy, however, because the electron beam is very

diffused in WRITE-THRU mode to decrease its intensity below that required for storage. This decreased intensity makes the beam difficult to see, especially if it is not properly adjusted (see the ADAPTS User's Guide).

```
 10  REM PROGRAM TO DISPLAY X IN WRITE-THRU MODE
 20  CALL PULSE, 63
 30  CALL CRT
 40  CALL SIZE, 2
 50  CALL POS, 0, 0, 0
 60  PRINT 'X'
 70  CALL STATUS, 1, A
 75  IF A=0 THEN 50
 80  CALL PULSE, 64
 90  CALL TTY
100  END
```

In the example above, the program will terminate normally when STATUS line 1 is grounded. Otherwise the "x" will be repetitively written in the center of the screen in WRITE-THRU mode.

The programmer may wish to erase the screen without changing the beam position. He may do this by using the command

```
CALL PULSE, 60
```

However, the screen will not be checked for ERASE INTERVAL true after erase, as it would be if the CALL ERASE command were used. STATUS line 57 may be checked to determine when the ERASE INTERVAL is over (true condition) so that the program may not write until the screen is in a ready-to-write state. For example:

```
10 CALL PULSE, 60
20 CALL STATUS, 57, A
25 IF A=0 THEN 20
30 REM CONTINUE WITH PROG IF STATUS TRUE
. . . .
```

The example on the following page summarizes many of the principles of operating the oscilloscope display. When RUN is typed, the program will type

```
GIVE FREQUENCY IN kHz?
```

Typical values might be in range 1-10. The program will display a sine wave of this frequency on the oscilloscope screen. Then it will ask the user for scale factors in X and Y. A scale factor larger than 100 percent will decrease the number of actual units per inch; this will seemingly decrease frequency (X scale factor) and increase the amplitude (Y scale factor).

Note from line 290 that the display will always extend from the extreme left of the screen to the extreme right no matter what scale factor is selected. The amplitude in the Y

direction is ± 200 for a scale factor of 100 percent and will vary according to the input accepted by line 140.

Function FNA(X) is used to convert the X axis to a time scale which, for a scale factor of 100, runs from − 0.5 millisecond to + 0.5 millisecond. Therefore, the frequency in kHz will be the number of actual cycles displayed on the screen (before scale factors are introduced). Line 230 defines a function FNB(X) which is the scale factor times the amplitude at 100 percent times sine (2 pi ft.). The data points are printed as small x's. Note that if the Y scale factor is made larger than 200 percent any data point which the program trys to print below the page will cause the display to halt, waiting for the go-ahead signal (any key) to be given before erasing the page and continuing the display.

```
 LIST
 10 CALL   TTY
 20 PRINT
 30 PRINT 'GIVE FREQUENCY IN KHZ';
 40 INPUT F
 50 READ X1,X2,Y1,Y2,P
 60 READ S1,S2
 70 DATA -512, 511, 200,-200
 80 DATA 200,  100,  100
 90 GOSUB 250
100 PRINT 'GIVE SCALE FACTORS IN % '
110 PRINT 'X SCALE FACTOR =';
120 INPUT S1
130 PRINT 'Y SCALE FACTOR =';
140 INPUT S2
150 GOSUB 250
160 PRINT 'TYPE 1 TO REPEAT, 0 TO END PROGRAM';
170 INPUT R
180 IF R= 0 THEN 240
190 RESTORE
200 IF R= 1 THEN 10
210 IF R# 1 THEN 160
220 DEFFN A(X)=( 100*X/S1)/((X2-X1)* 1000)
230 DEFFN B(X)=(S2*Y1/ 100)*SIN( 6.27999* 1000*F*FNA(X))
240 END
250 REM SUBROUTINE
260 CALL ERASE
270 CALL  SIZE, 1
280 CALL   CRT
290 FOR X=X1 TO X2 STEP (X2-X1)/P
300 CALL    POS,X,FNB(X), 0
310 PRINT 'X'
330 CALL   SIZE, 2
340 CALL    TTY
350 RETURN
```

INFORMATION DISPLAY ON KEYBOARD OSCILLOSCOPE DISPLAY

This unit, Model A-930, is used in lieu of the A-620/73. It is a free-standing, pedestal mounted, keyboard-CRT unit. In an ADAPTS system, its function is almost identical to that of the A-620/73-Teletype keyboard combination (see preceding section).

The commands given in table 7-12 apply identically to keyboard CRT with the following exceptions:

a. With the ORIGIN set to (0,0) the screen is organized so that:

$0 \leq X \leq 1023$
$0 \leq Y \leq 780$

b. The following command puts the A-930 on-line:

CALL KBCRT • Switches alphanumeric output stream to the keyboard-CRT

c. Just one character size is available on the A-930. The SIZE command controls the " interline" spacing:

CALL SIZE,S • Sets the interline spacing to (11 points) * (S), where (S) may be a formula, but must evaluate to a positive integer between 1 and 80. Once set, SIZE remains unchanged until altered by another SIZE or ZOOM command. Initially, S = 2, which allows 36 lines of 74 characters each to be written on the screen. The ZOOM command affects SIZE as follows:

$$\text{new size} = \text{greatest integer} \left(\frac{S * \text{ZOOM factor}}{100} \right)$$

d. RETURN KEY • Pressing the RETURN key on the CRT keyboard (anytime after the CALL KBCRT command) positions the beam to:

X = 0
Y = unchanged

e. CTRL,FORM keys • Pressing the holding the CTRL key, then pressing the FORM key erases the CRT and positions the beam to:

X = 0
Y = 780

(continued)

f. LINE FEED key

- Pressing the LINE FEED key positions the beam to:

  X = unchanged

  Y = current Y - 11 * S

  If the new Y< 0 the speaker tone sounds continuously until any key is pressed which executes a CTRL,FORM

g. The auxiliary commands listed in table 7-12 do not apply to the A-930.

INFORMATION OUTPUT ON DIGITAL X-Y PLOTTER

A summary of the commands for operating the plotter is given in table 7-13. Note the similarity between these commands and those for the oscilloscope display units. This similarity allows the same EBASIC program to output alphagraphic information to the CRT or the plotter. Plotter commands use paramenters which may or may not be used by the CRT. Some parameters are also optional for the plotter. Nevertheless the plotter and CRT software packages each recognize their own valid parameters and function accordingly.

Table 7-13. EBASIC Digital X-Y Plotter Commands

Command	Function
CALL PLOTTR	Switches alphanumeric output stream to the plotter
CALL TTY	Switches alphanumeric output stream to the Teletype
CALL INIT	Initializes the plotter by raising the pen and returning it to the load point (lower left corner); the origin is reset to (0,0), the character size to (2), the scale factors to (100), and the zoom factor to (100). This establishes an X and Y reference zero point and locates the pen point so that a new chart may be loaded without interference.
CALL ERASE	Raises the pen and returns it to the load point (lower left corner of the page).
CALL SIZE,S,OR	Sets the size (S) and orientation (OR) of alphanumeric characters. (S) may be a formula, but must evaluate to a positive integer between 1 and 16. Initially, (S) = 2 which produces characters about 1/8 inch high. (OR) is optional; if not used it is set to zero; (OR) may be a

(continued)

Table 7-13. EBASIC Digital X-Y Plotter Commands *(continued)*

Command	Function
	formula, but must evaluate to a positive integer between 0 and 3 as follows: if 0, upright 1, turned right 2, upside down 3, turned left The ZOOM command affects SIZE as follows: $\text{new size} = \text{greatest integer} \left(\frac{S * \text{ZOOM factor}}{100} \right)$
CALL SCALE, SF1,SF2	Sets the scale factors (SF1 for X) and (SF2 for Y) to enlarge or reduce graphic output only while alphanumerics simply change absolute position. Once set, SCALE remains unchanged until altered by another SCALE or ZOOM command. Initially, both (SF1) and (SF2) equal 100; (SF2) is optional and if omitted, is set to SF1. The ZOOM command effects SCALE as follows: $\text{new scale} = \text{greatest integer} \left(\frac{\text{SF1,SF2} * \text{ZOOM factor}}{100} \right)$ (SF1) and (SF2) can be formulas, but must evaluate within the range −32768 to 32767.
CALL ZOOM, ZF	Sets the zoom factor (ZF) to enlarge or reduce the entire alpha-graphic presentation. Once set, ZOOM remains unchanged until altered by another ZOOM command. Initially, ZF = 100; it can be a formula, but must evaluate within the range −32768 to 32767.
CALL ORG,X,Y,M	Stores bias values for X and Y coordinates permitting translation of the origin of the coordinate system to any desired point. Once set, the bias values remain unchanged until altered by another ORG command. Initial bias values are both zero. (X) and (Y) can be formulas, but must evaluate within the range −32768 to 32767. (M) must evaluate to zero (specifying absolute) or one (relative).
CALL POS,X,Y,M	POS raises the pen and repositions it to X,Y;

(continued)

Table 7-13. EBASIC Digital X-Y Plotter Commands *(continued)*

Command	Function
CALL POINT,X, Y,M	POINT raises the pen, repositions it to X,Y, and then lowers it momentarily to make a small dot on the page. Repositioning is to the new X- and Y- coordinates absolute (if M = 0) or relative (if M = 1).
CALL VECT,X, Y,M	Draws a straight line from the present pen position to the new X- and Y- coordinates absolute (if M = 0) or relative (if M = 1).
CTRL,FORM keys	Pressing and holding the CTRL key on the Teletype, then pressing the FORM key raises the pen and returns it to the load point (lower left corner).
RETURN and LINE FEED keys	Pressing these keys on the Teletype affects the beam position of CRTs, but are ignored for the plotter.

UTILITY SUBROUTINES

The RENUMB subroutine permits the user to renumber EBASIC program statements. The command format is:

```
CALL RENUMB, line number, increment
```

which renumbers the entire current, in-core EBASIC program. The first statement of the renumbered program is labeled (line number) and each succeeding statement is labeled (previous line number) + (increment). In addition, line numbers referenced in all EBASIC statements (except CALL arguments, and REM statements) are automatically adjusted to match the new numbers. Line numbers must be in the range zero to 9999. Note the use of the RENUMB command in the following example:

```
   LIST
  1    LET A = 2
 11    GOTO 100
 12    PRINT 'ERROR IF MESSAGE TYPED'
 50    IF A = 1 THEN 124
 60    STOP
100    CALL STATUS, 1, R
110    IF R = 0 THEN 100
122    GOTO 50
124    END
```

(continued)

```
        CALL RENUMB,10,5
        LIST
10      LET A = 2
15      GOTO 35
20      PRINT 'ERROR IF MESSAGE TYPED'
25      IF A = 1 THEN 50
30      STOP
35      CALL STATUS,1,R
40      IF R = 0 THEN 35
45      GOTO 25
50      END
```

Report Program Generator IV (RPG IV)

TABLE OF CONTENTS

SECTION 1 - INTRODUCTION

The **RPG (Report Program Generator) IV language** is an advanced version of the widely used RPG commercial and general data-processing systems. RPG IV permits the concise coding of powerful programs in a simple and efficient manner. Thus, users with backgrounds other than data processing can use RPG IV problem-solving techniques without extensive training or practice.

RPG IV improves on basic RPG in that it incorporates many automatic features and powerful procedural statements. RPG IV is particularly adapted to processing data for the output of reports, but has many other applications as well.

This manual is divided into five sections.

Section 1 introduces RPG IV by explaining its basic features and illustrating them with a simple example.

Section 2 gives system concepts on hardware and software.

Section 3 details the RPG IV language. It explains the types of statements and their coding, and gives the format and use of each type of statement in the language.

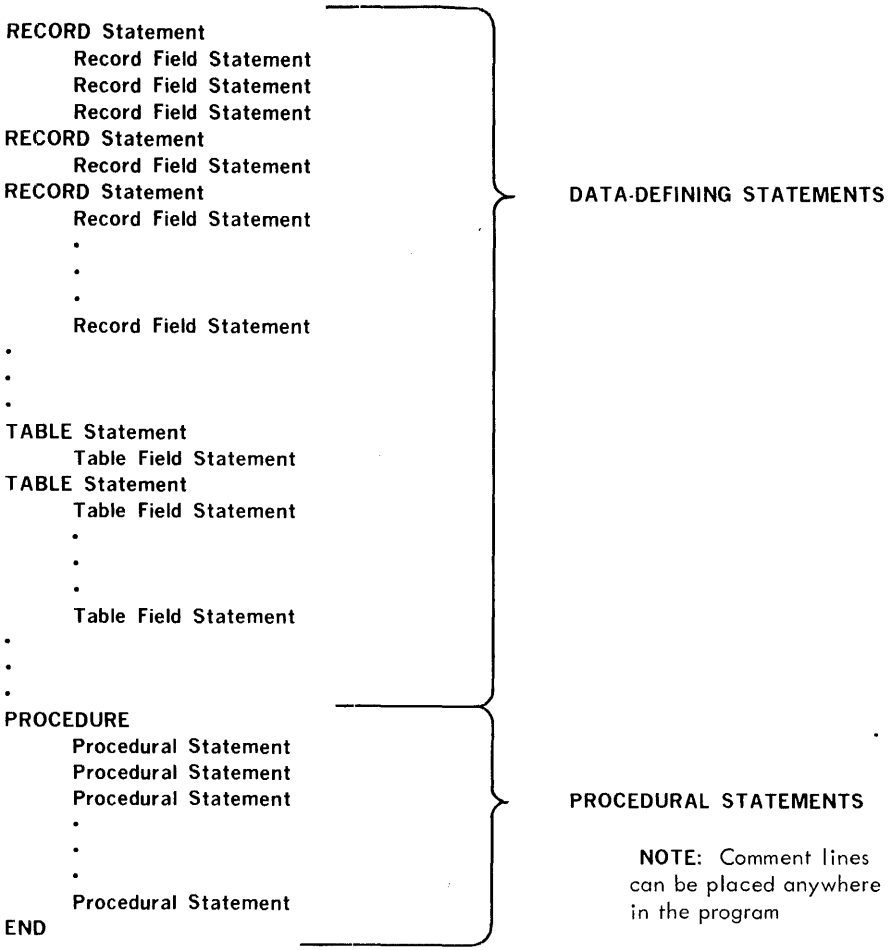
Section 4 explains the use of the hardware in running RPG IV programs.

Section 5 gives an advanced sample program. This program illustrates uses of the instructional material given in previous chapters.

This manual is written for the user, who may or may not have programming experience, and explains the use of RPG IV so that nonprogrammers can apply the language to problem-solving. No special training or experience is required, except for section 4, which is written for the computer operator. However, this chapter is an explanation of how to run completed programs on the computer and does not impair the ability of the nonprogrammer or nonoperator to write the programs according to the rules given in the other chapters.

STRUCTURE OF RPG IV PROGRAMS

RPG IV programs comprise two sequences of statements. First, there is a sequence of **data-defining statements** that defines the structures and formats of the data to be processed. This is followed by a sequence of **procedural statements** that processes the data through the structures defined in the first sequence. This processing yields the output in the desired form. Figure 1-1 shows the general layout of a typical program in RPG IV.



VTII-1009A

Figure 1-1. Layout of a Typical RPG IV Program

These two sequences of statements handle tables and records, update files, produce reports, and can deal with any other business-oriented applications. Section 3 details the data-defining and procedural statements, and gives their individual formats and uses.

In addition to the program itself, there is the data to be processed. This is a separate sequence of input that has the format(s) specified by the data-defining statements of the program.

Simple RPG IV programs with explanations are provided later in this section.

STATEMENTS

A **statement**, whether it is a data-defining or a procedural statement, is one line of information written in the RPG IV language. It consists of 80 characters corresponding to the 80 columns of a standard punched card.

RPG IV ignores columns 70 through 80. Use these columns for statement identification, comments, or programming aids as desired.

NOTE

The data used with the program can be in any column, following the specifications of the data-defining statements.

The statement itself is in columns 1 through 69. It comprises elements defined in the following section arranged in a format that depends on the individual statement (section 3). Regardless of the elements contained in a statement or the format requirements within a statement, each statement is freeform, i.e., there are no requirements for spacing, indentation, or column use within columns 1 through 69. You need no special coding forms.

DATA-DEFINING STATEMENTS

As indicated in figure 1-1, there are four types of data-defining statements that provide a definition of the data structures to be used by the program. These data structures are records and tables. **Records** hold intermediate results and data being input from or output to files. **Tables** contain related, repetitive data items.

Both records and tables are divided into fields. **Fields** are the elementary variables of any RPG IV program. The computations performed by the program, its logic, and its final output are based on the manipulation of fields and their contents.

A **record statement** identifies a record and specifies the conditions under which this record is manipulated.

A **record field statement** identifies and defines all of the fields in the record. All record field statements pertaining to a given record immediately follow the record statement for that record.

A **table statement** identifies a table and specifies its size.

A **table field statement** identifies and defines all of the fields in the entries in a table. All table field statements pertaining to a given table immediately follow the table statement for that table. Each entry in a given table has the same field structure as any other entry in that table.

The formats, elements, and uses of data-defining statements are explained in section 3.

PROCEDURAL STATEMENTS

As indicated in figure 1-1, procedural statements follow the data-defining statements. They direct the execution of the program as it processes the data previously defined by the data-defining statements.

The **PROCEDURE** statement is the first procedural statement. It is not executable, but merely serves as the divider between the data-defining statements and the procedural statements. It terminates the processing of data definitions and begins the processing of procedural manipulations. This statement has only one form: the single word PROCEDURE.

Subsequent procedural statements manipulate the data to obtain the desired output. Their formats, elements, and uses are explained in section 3.

Procedural statements are executed in the order of their appearance in the program unless the specified condition is not met, or unless the program is directed to another statement by a GO TO or PERFORM statement (section 3).

The **END** statement is the last procedural statement. It is not executed, but merely serves as a signal that the program is finished. This statement has only one form: the single word END. It is the last statement in the program.

ELEMENTS OF RPG IV STATEMENTS

RPG IV statements contain combinations of the following elements arranged in formats given in section 3:

a. Statement numbers

b. Names

c. Conditions

d. Indicators

e. Constants

f. Literals

g. Expressions

h. Comments

The elements are defined in this section and their uses illustrated in the sample program (section 3) in giving the format of each RPG IV statement, exhaustively explains the application of these elements in the statements.

STATEMENT NUMBERS

A **statement number** from 1 to 9999 can begin any procedural statement. It identifies the statement so that the program has access to it as required (e.g., in program loops, jumps, and conditional processing). Statements that do not require other than sequential access need not be numbered.

NAMES

A **name** identifies data or a subroutine referenced by the program. It comprises one to six alphanumeric characters (numbers and letters), the first of which is alphabetic. No blanks are allowed.

Examples:

```
A
X15
FIELD3
J7U5
```

A **record name** identifies an area of memory that provides space for the characters comprising the record. A record name is assigned by a RECORD statement (section 3).

A **table name** identifies an area of memory that provides space for a series of data entries, i.e., a table. All entries in a given table have identical field assignments. A table name is assigned by a TABLE statement (section 3).

A **field name** identifies a contiguous set of character positions in a table entry or record. It is assigned by a FIELD statement (section 3).

An **implied field name** identifies a field not named in a FIELD statement. The appearance of an implied field name in a procedural statement causes assignment of a memory area to it. This area is large enough to hold any character string or number entered.

A **subroutine name** identifies a special procedure outside the program. This procedure (subroutine) performs one of many special functions.

The above types of names conform to the format given at the beginning of this subsection. In addition, there are qualified and subscripted names that take modified formats.

A **qualified name** identifies a field and its record or table so that it is not confused with a like-named field in another record or table. A qualified name consists of a record or table name, a period, and a field name. No blanks are allowed.

Examples:

```
UPDATE.HOURS
OUTPUT.HOURS
C45Y.Y7
J289RR.Y7
```

A **subscripted name** identifies a table entry. It consists of a table name followed by a computational expression (see EXPRESSIONS) in parentheses. No blanks are allowed. The integral portion of the result of the computation is the number of the table entry referenced.

Examples:

```
ACCNT(INDEX)
OUTPUT.NAME(3)
VALUE7(X+3)
NORTH.Y3(X*2.53)
```

Note: Up to 1023 names (explicit and implied) are allowed.

CONDITIONS

A **condition** can be imposed on any procedural statement by placing the condition in parentheses in front of the statement. Such a statement is executed only if the condition is met when the statement comes up for execution.

Example:

```
(TIME=10) MOVE INPUT.JOB,OUTPUT.JOB
```

is a MOVE statement (section 3) that moves the contents of the field JOB in record INPUT to the same field in record OUTPUT only if TIME has the value 10 when the program is ready to execute this instruction. Time is defined in the program prior to this statement.

If the statement is numbered, the condition follows the number.

Example:

```
100 (TIME=10) MOVE INPUT.JOB,OUTPUT.JOB
```

INDICATORS

An **indicator** is a program switch that can be on or off. By program switch we mean that the program itself, not part of the computer hardware, turns the switch on or off. You can thus use these switches to control the operations performed by your program, and specify the conditions under which these controls are activated. For instance, you can specify that certain statements be executed if, and only if, certain indicators are on or off.

RPG IV has three types of indicator: general, control break, and special. The indicators and their uses are given in appendix A for later referencing convenience. Refer to this appendix as you study this section.

A **general indicator** is turned on or off by a general procedural statement (section 3). There are 99 general indicators, specified in coding by the symbols #1 through #99. Used in procedural statements, they specify conditions to be met for the execution of any part of the program. For example:

```
(#2) ENTER INPUT,TABLEA
```

enters INPUT in TABLEA only if general indicator #2 is on; and

```
(#1 AND NOT #65) ENTER INPUT,TABLEA
```

enters INPUT in TABLEA only if general indicator #1 is on *and* general indicator #65 is off (the conditions under which these indicators are on or off will have previously been specified by your general procedural statements). All general indicators are off until you write a general procedural statement that turns them on. Of course, you can turn them off again with other procedural statements.

A **control break indicator** is used in the direct updating of a record control field and as a condition for statement execution. There are ten levels of control break: #C1 through #C10. the uses of these indicators are explained in section 3.

A **special indicator** is used like a general indicator except that the conditions are implicit in the indicator itself and are not otherwise specified. A special indicator is turned on or off by a general procedural statement, or by one of the explicit procedural statements SET ON, SET OFF, or SET conditional (section 3). For example:

```
(#") ENTER INPUT,TABLEA
```

enters INPUT in TABLEA only if the previous statement was executed. One frequent use of #" is to repeat long or complicated conditions for successive statements.

NOTE

All special indicators except #OV are off until a statement that turns them on is executed. However, because #OV is normally used to begin a new report page, it is initially on.

CONSTANTS

A **constant** is a positive or negative number used by the program. It can contain up to 14 significant decimal digits before, and nine after, the decimal point. For integer constants, no decimal point is required. No blanks are allowed.

Examples:

```
375.125
100
.0333333
12345678901234.123456789
-123
```

LITERALS

A **literal** is a string of alphanumeric characters (on one line) used by the program. A literal is enclosed within apostrophes. Blanks are allowed in literals.

NOTE

If an apostrophe is to be part of a literal, use two consecutive apostrophes.

Examples:

```
'THIS IS A LITERAL'
'DON''T'
'$350.00'
```

EXPRESSIONS

An **expression** in RPG IV is one of three types: computational, relational, or conditional.

A **computational expression** is a combination of constants and/or numeric fields with the arithmetic operators + (addition), - (subtraction), * (multiplication), and / (division). In a computational expression, operations within parentheses are performed first, and multiplication and division are performed before addition and subtraction. Within these levels, operations are performed from left to right. After nine digits to the right of the decimal place in the result of the computation, there is rounding for multiplication and division, and truncation for addition and subtraction.

Examples:

```
A+B*C
```

multiplies B by C and adds A;

```
(X*Y)+(U*V)
```

multiplies X by Y and U by V, and adds the results;

```
-VAL1(J)/37.5
```

divides VAL1(J) by 37.5 and negates the result; and

```
-(A*(B+C+D+E))
```

multiplies A by the sum of B, C, D, and E, and negates the result. Numeric fields in a computational expression can contain editing characters and embedded blanks without affecting the arithmetic interpretation since only the digits, sign, and decimal point are significant. Thus, a field with two implied decimal places could contain either 00002575 or $**25.75 and be interpreted as 25.75 in the computation.

A **relational expression** compares two computational expressions, or two alphanumeric fields or literals, for a specific relational condition. The two expressions, fields, or literals are separated by one or two of the relational operators <, =, and >, as follows:

<	Less than
>	Greater than
=	Equal to
< = or =<	Not greater than (less than or equal to)
> = or =>	Not less then (greater than or equal to)
<> or ><	Not equal to (less than or greater than)

If the relation is true, the condition is met. If not, the condition is not met.

Examples:

```
FIELDA<=FIELDB
```

states that the condition is met when FIELDA is less than or equal to FIELDB;

```
A*B>10
```

states that the condition is met when A*B is greater than 10; and

```
A1(INDEX) >< LIMIT
```

states that the condition is met when A1(INDEX) is not equal to LIMIT. Note that a constant or implied numeric field cannot be compared with an alphanumeric field or literal.

A **conditional expression** combines indicators and/or relational expressions with the logical operators AND, OR, and NOT to form a logical condition. In conditional expressions, operations within parentheses are performed first, and NOTing is performed before ANDing, which is performed before ORing. Within these levels, operations are performed from left to right. NOT can follow another logical operator. At least one blank follows each logical operator if the next character of the statement is a letter or digit.

Examples:

```
#1 AND NOT #2
```

states that the condition is met only if indicator #1 is on *and* indicator #2 is off;

```
NOT A>B OR #OV
```

states that the condition is met only if A is not greater than B *or* if the page overflow indicator #OV is on.

COMMENT LINES

A **comment line** improves the format of the listing or documents the program. It appears in the listing but neither acts nor is acted upon. A comment line is either entirely blank or has an asterisk as the first nonblank character. Blanks are allowed.

Examples:

```
*REMARKS CAN OFTEN CLARIFY A PROGRAM
          *THIS IS A COMMENT $85+N
```

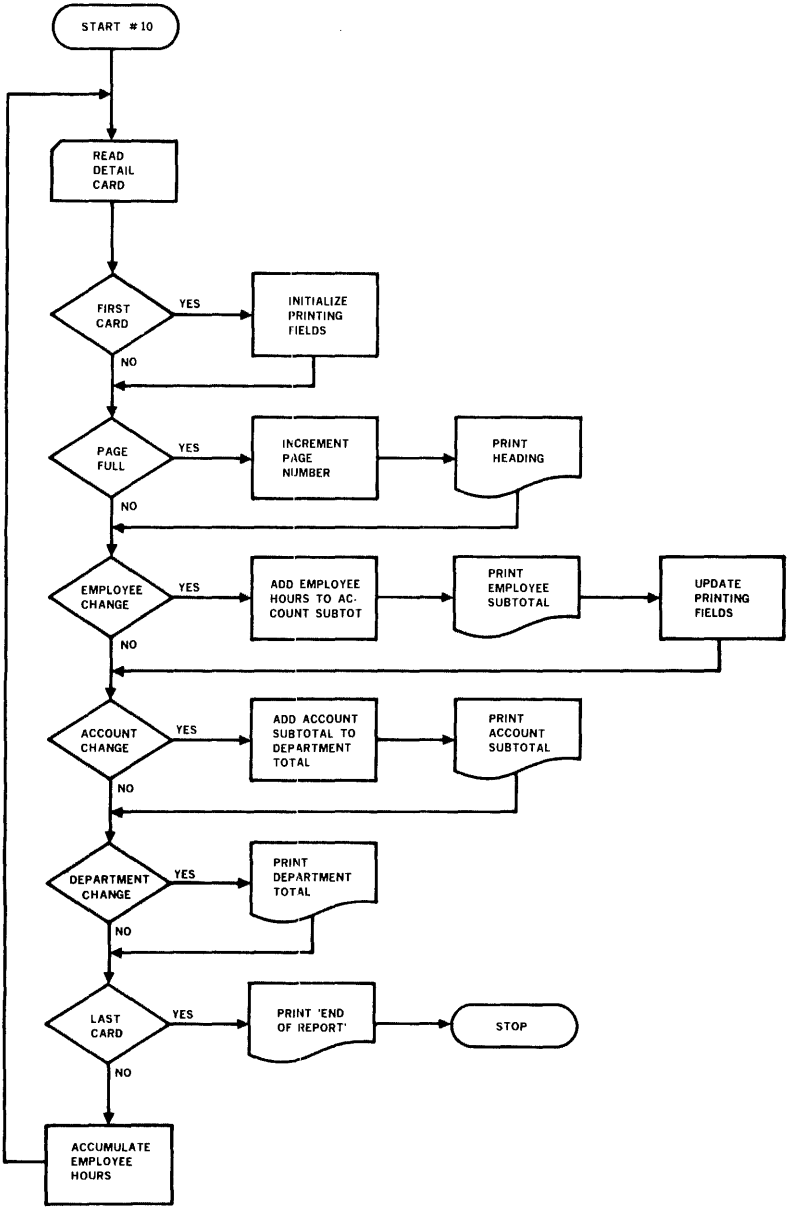
SAMPLE PROGRAMS

This section shows a flowchart (figure 1-2), an RPG IV program (figure 1-3), the data to be processed by the sample program (figure 1-4), and the resulting report (figure 1-5). It would serve little purpose here to give a detailed description of the total operational sequence involved in this data processing, since this sample is intended to serve as a reference guide while reading the material in sections 2 and 3, and to show how the program, data, and results are related.

Notice that the first data-defining statements are a group of literals under the record named HEAD. The first procedural statement reads a card. If it is the first **control break** (#F on), the output control fields are initialized by the move statement. If this card would cause page overflow on printing, the printer goes to the top of the next page, increments the page NO, and prints the heading HEAD (the overflow indicator #OV is also on at the beginning of a program since it is assumed that the first line of output will be at the top of the page). Thus, the first printer action, PRINT if #OV is on, skips to the top of the next page ($C1), prints HEAD, and skips a line ($A1). Note that the record named HEAD is printed as is, with DEPARTMENT in character positions 16 through 25. ACCOUNT NUMBER in 29 through 42, EMPLOYEE in 47 through 54, HOURS in 59 through 63, PAGE in 72 through 75, and the page number (i.e., the value of NO) in 77 and 78 as specified in the data-defining statements under the RECORD HEAD card.

Now that the heading is printed, the program considers the card already read. Since the READ CARD procedural statement specifies the record named INPUT, the data on the card are placed into the record INPUT according to the format given by the data-defining statements of that record. Thus, the record field DEPT comprises character positions 1 through 3 of the record, EMPNO 4 through 9, ACCT 10 through 24, and HOURS 75 through 80. The designation 80.1 specifies that the record field HOURS has one place to the right of an implied decimal point. Examination of the first data card shows that the data are in compliance with the specifications of the record named INPUT. Thus, 10A in card columns 1 through 3 is the department number DEPT, 210356 in card columns 4 through 9 is the employee number EMPNO, etc. Upon printing, the positions of these fields shift to those in the record named OUTPUT so that the department number is printed in character positions 20 through 22, etc. The symbols #C1, #C2, and #C3 are control break indicators that in this case specify that an employee number be printed for every entry, but that account and department numbers be printed only for the first applicable entry.

Further details of the procedure will become apparent on further study and the reading of sections 2 and 3.



VTI2-0359

Figure 1-2. Flowchart for the Sample RPG IV Program

```
VARIAN RPG IV SOURCE LISTING

     *
     *              SAMPLE RPG IV PROGRAM
     *
                    RECORD HEAD
                      (16,25)     'DEPARTMENT'
                      (29,42)     'ACCOUNT NUMBER'
                      (47,54)     'EMPLOYEE'
                      (59,63)     'HOURS'
                      (72,75)     'PAGE'
                      NO(77, 78.0) Z
                    RECORD INPUT
                      DEPT(1,3) #C3
                      EMPNO(4,9) #C1
                      ACCT(10,24) #C2
                      HOURS(75,80.1)
                    RECORD OUTPUT
                      DEPT(20,22) B,P#C3
                      ACCT(29,41) B,P#C2
                      EMPNO(48,53) P#C1
                      EHOURS(58,63.1) B,Z,D
                    RECORD SUBTOT
                      AHOURS(66,71.1) B,Z,D
                    RECORD TOTAL
                      DHOURS(73,78.1) B,Z,D
                    PROCEDURE
     10             READ CARD INPUT
       (#F)         MOVE INPUT.DEPT,OUTPUT.DEPT
       (#")          "    INPUT.ACCT,OUTPUT.ACCT
       (#")          "    INPUT.EMPNO,OUTPUT.EMPNO
       (#OV)        COMPUTE NO=NO+1
       (#")         PRINT $C1,HEAD,$A1
       (#C1)        ADD EHOURS,AHOURS
       (#")         PRINT OUTPUT
       (#")         POST INPUT,OUTPUT
       (#C2)        ADD AHOURS,DHOURS
       (#")         PRINT SUBTOT
       (#C3)        PRINT TOTAL
       (#LC)        STOP 'END OF REPORT'
                    ADD HOURS,EHOURS
                    GO TO 10
                    END
```

VTI1-1013B

Figure 1-3. Sample RPG IV Program

```
10A210356ALPHA-29107                                    1200
10A210356ALPHA-29107                                     800
10A350017ALPHA-29107                                     392
10A350017ALPHA-29107                                    1500
10A151179BETA-35                                         800
10A161711BETA-35                                         800
10A290238BETA-35                                         800
10A750192BETA-35                                         800
25B019372HOUSE-1997                                      505
25B019372HOUSE-1997                                      405
25B019372HOUSE-1997                                       50
25B317911HOUSE-1997                                     1500
25B607712HOUSE-1997                                     1200
```

VTI1-1014A

Figure 1-4. Data for the Sample RPG IV Program

```
DEPARTMENT    ACCOUNT NUMBER   EMPLOYEE HOURS     PAGE

     10A      ALPHA-29107      210356   200.0
                               350017   189.2
                                        389.2
              BETA-35          151179    80.0
                               161711    80.0
                               290238    80.0
                               750192    80.0
                                        320.0
                                             709.2
     25B      HOUSE-1997       019372    96.0
                               317911   150.0
                               607712   120.0
                                        366.0
                                             366.0
END OF REPORT
```

VTII-1015

Figure 1-5. Final Report from Sample RPG IV Program

Four types of RPG IV coding forms provide convenient documentation and organization for programs.

SECTION 2 - SYSTEM CONCEPTS

HARDWARE

The Varian 73/620 RPG IV System can operate in one of three environments: as a stand-alone system, under the master operating system (MOS), or under the VORTEX system.

The stand-alone version is a card-oriented system designed for a minimum hardware configuration consisting of a Varian computer with 4,096 words (4K) or more of memory, a card reader (620-25), a card punch (620-27), and a line printer (620-77).

The MOS version of RPG IV operates on any standard MOS configuration with 12K or more of memory and utilizes the I/O-device-independence inherent in MOS. The minimum MOS to operate RPG IV would be a 12K computer, a magnetic tape unit, and a Teletype. Expanded configurations are supported and include multiple magnetic tape units, card devices, line printers, and rotating memory devices.

The VORTEX version of RPG IV operates on any standard VORTEX configuration with a background partition size of at least 6K. It is also device-independent.

For additional information on the hardware system, refer to the Varian documentation on the 620 and 73 computers and the individual peripherals. Section 4 explains how RPG IV programs are complied and run in the different environments.

SOFTWARE

You provide two pieces of software to produce reports on the Varian RPG IV system:

a. You write a **program** according to the directions in section 3. This program comprises two parts: data-defining statements to specify the forms that the data to be processed will take, and procedure statements to specify how the data in the defined forms will be processed.

b. You supply **data** according to the specifications given in the data-defining portion of your RPG IV program.

The software supplied with the *Varian RPG IV System* processes the data you have supplied according to the specifications of the program you have written.

The basic software component of the system is the **RPG IV two-part compiler**. This component compiles your program and yields an **object deck**. The object deck, the **RPG loader**, and the **RPG runtime** support program process your data. The use of these software components is explained in section 4. All except the object deck are supplied with the *Varian RPG IV System*. (The object deck, of course, is the output of the compilation of your program by the supplied RPG IV two-part compiler.)

SECTION 3 - RPG IV STATEMENTS

As explained in section 1, there are two general types of statement in RPG IV. This section gives the specifications for data-defining statements and procedural statements.

In the descriptions of the statement formats, **boldface** type designates required items and *italic* type designates optional items. Items in capital letters are coded just as written. Items in lower-case letters represent variables, constants, values, etc.

DATA-DEFINING STATEMENTS

Data-defining statements are the first statements in an RPG IV program. They provide a definition of the data structures (records and tables) to be used by the program and processed according to the procedural statements.

The data-defining statements are divided into *record statements, record field statements, table* statements, and *table field statements.*

A **record statement** identifies a record and specifies the condition under which this record is manipulated. It is followed by the **record field statements** pertaining to the fields of this record, identifying and defining the fields. A **table statement** identifies a table and specifies its size. It is followed by the **table field statements** pertaining to the fields of the entries in this table, identifying and defining them.

RECORD STATEMENT

A **record statement** identifies a program record and its area in memory. It contains identifiers that specify the selection criteria for data to be input to the record by a READ CARD procedural statement and an indicator that is turned on when data are input to the record and turned off when it is not. The format of a record statement is

RECORD name *(identifier,identifier,...)indicator*

where **name** identifies the record and its area in memory, *identifier* is a record selection code, and *indicator* is a symbol for an indicator that is turned on when data are accepted and input into the record or off when the data are rejected.

For example, the record statement

```
RECORD GAIN (80C1)#17
```

identifies the record named GAIN, specifies that a READ CARD statement can enter data in this record only when column 80 of the data card contains a one, and turns indicator #17 on if data are accepted or off if data are rejected.

If the record statement contains no selection criterion *(identifier)*, any READ CARD statement that references this record will input data to the record.

Identifier

The **identifier** in a record statement is a record selection code that specifies the criteria for the input of data to this record by a read card statement. If the criteria specified by the identifier are met, the data from the card are input to the specified record and the indicator designated in the RECORD statement is turned on. Identifiers are enclosed in parentheses.

An identifier has one of the following formats:

```
pCx  pDx  pZx  pNCx pNDx pNZx
```

where C, D, and Z specify that the selection is based on the entire character (C), the digit (0-9 punches) portion (D), or the zone (11-12 punches) portion (Z); p is a number from one to 80 specifying the card column used for comparison; x is a character punched in that column; and N (NOT) specifies that the comparison must fail for data to be accepted into the record.

For example, the identifier 1C3 specifies that the character in column 1 of the data card must be a three for data to be accepted into the record. Identifier 80NZA specifies that for data to be accepted into the record, the character in column 80 of the data card *cannot* have the zone bits of the character A (i.e., since A has a 12-punch as a zone bit, the character in column 80 must have a different zone bit configuration than a 12-punch).

Multiple Identifiers

Record statements can contain multiple identifiers to specify ANDed or ORed conditions for the acceptance of data into the record.

A sequence of identifiers separated by commas and included within one set of parentheses ANDs the selection criteria of the individual identifiers. For example, the sequence

```
(15CX,20NZ-)#79
```

specifies that for acceptance of data into the record, there is an X in column 15 *and* there cannot be the zone bits of the minus sign in column 20. The indicator is given at the conclusion of the identifier sequence.

A sequence of identifiers in separate subsequences, separated by commas and indicator symbols, ORs the selection criteria of the individual identifiers. For example, the sequence

```
(80C1)#17,(80C2)#27
```

specifies that for acceptance of data into the record, column 80 contains a one *or* a two. The sequence

```
(22D$)#40,(54CE)#40
```

Specifies that for acceptance of data into the record, column 22 contains the numeric bits of the $ (i.e., a 3-8 punch) *or* column 54 contains an E. An indicator is given after each identifier.

These specifications can be combined. For example, the sequence

```
(1Z-)#1,(1NZ-,2C )#2
```

specifies that for acceptance of data into the record, column 1 contains the zone bits of the minus sign, **or** if column 1 does not contain the zone bits of the minus sign *and* column 2 is blank.

RECORD FIELD STATEMENTS

All **record field statements** for a given record directly follow the record statement. They define all fields in the record. The fields can appear in any order and can overlap.

The *record field statement for an alphanumeric field* has the format

field (**first,last**),*parameter,parameter...*

where *field* is the name (if any) of the field, **first** is the number of the first character position in the field, **last** is the number of the last character position in the field, and *parameter* is one of the parameters discussed in a later subsection.

Statement-identifying names and logical operators (e.g., RECORD, MOVE, NOT) cannot be used as record field names.

If more than one parameter is required, enter additional parameters, separated by commas, following the first ***parameter***. Parameters can appear in any order.

The *record field statement for a numeric field* has the format

field (**first,last.decimal**),*parameter,parameter,...*

where the definitions are as above except that **last** is followed by a decimal point and **decimal**, which specifies the number of digits to the right of the implied decimal point in

the field. It is present for every numeric field, even if the value is zero. If the field contains an actual decimal point, its position overrides the specification in the record field statement.

An example of a record field statement for an alphanumeric field is

```
(16,25),'DEPARTMENT'
```

which places the literal DEPARTMENT in character positions 16 through 25 of the record to which the record field statement applies.

An example of a record field statement for a numeric field is

```
EHOURS (59,64.1)
```

which places data having one place to the right of the decimal point in the field EHOURS. This field occupies character positions 59 through 64 of the record to which the record field statement applies.

Negative Numbers

Three methods of expressing negative numbers are recognized by RPG: minus sign, credit symbol, and minus overpunch.

MINUS SIGN (-)

A number may have an appended minus sign to express a negative value. Space in the field definition statement must be made for the minus sign. The sign may appear anywhere within the field, but must be the last character excluding blanks.

CREDIT SYMBOL

A number may have an appended credit symbol to express a negative value. Space in the field definition statement must be made for the credit symbol. This symbol may appear anywhere within the field, but must be the last non-blank character.

MINUS OVERPUNCH

A number may have a minus overpunch to express a negative value. The numeral which is overpunched with a minus must be the last number in the field and be right-justified.

NOTE

Since a zero with a negative overpunch is undefined (11-0), the character uparrow ↑ (12-7-8) is used to denote a negative number ending in zero. This is applicable to the MOS and VORTEX versions only.

The following table lists the graphic representation for overpunched digits.

digit	graphic
0	!
1	J
2	K
3	L
4	M
5	N
6	O
7	P
8	Q
9	R

Parameters

The following parameters can be used with record field statements to define more closely the format of the data. Parameters can appear in any order. Each parameter is preceded by a blank or comma.

BLANK (B PARAMETER)

The **B parameter** consists of the letter B. It indicates that the field is to be cleared (blanked) after the record is output with a PRINT or PUNCH statement. An example of a record field statement containing this parameter is

```
ACCNT (10,19),B
```

CONDITIONAL POSTING (P PARAMETER)

The **P parameter** consists of the letter P followed by an indicator that is on for conditional posting. If the indicator is off when a POST or COLLECT operation would normally modify this field, no modification occurs. An example of a record field statement containing this parameter is

```
ACCNT (10,19),P#C1
```

EDITING PARAMETERS

An **editing parameter** consists of one of the letters or symbols listed below. It edits a numeric field according to the corresponding explanation. More than one editing

parameter can be used in a record field statement, but each must be separated by blanks or commas.

H Half-round the low-order digit.

Z Suppress leading zeros.

D Insert actual decimal point.

C Insert commas.

$ Insert $ before first digit.

* Replace leading zeros with asterisks.

– Allow one position to the right for a minus sign.

CR Allow two positions to the right for the credit sign (CR).

Editing parameters do not apply to alphanumeric fields. Examples of record field statements containing editing parameters are

```
EHOURS (59,64.1),D
EHOURS (59,64.1),B,Z,D
BAL (35,44.2),$,*,C,D,CR
```

AUDITING PARAMETER

The **auditing parameter** consists of any combination of the letters A, N, and S followed (optionally) by any combination of the letters R, L, and J; followed (obligatory) by an indicator symbol. The auditing parameter checks the validity and positioning of characters in a field.

The first set of letters indicates the characters permitted by the audit:

A permits alphabetic characters

N permits numerals

S permits special characters

These can be combined. For instance, AN permits alphabetic characters and digits, but no special characters. The presence of a nonpermitted character causes the audit to fail.

The second set of letters, if used, indicates the allowed positions of the permitted characters:

R right-justified (rightmost character nonblank)

L left-justified (leftmost character nonblank)

J all characters juxtaposed (no embedded blanks)

If none of these letters appears in the auditing parameter, all positions are permitted for characters and blanks.

The indicator following the letters is turned on if the audit passes and off if the audit fails. For example, the auditing parameter NRJ#20 turns indicator #20 on only if the rightmost characters are digits and there are no nonnumeric characters or embedded blanks (leading blanks are allowed).

Special case: The letter J alone with an indicator symbol causes an audit for an all-blank field.

Examples of record field statements containing auditing parameters are:

```
NAME (21,30),AL,#40
 ZAP (1,69),J#1
 NUMBER (1,10),N#99
 CLOSED (11,27),ANSLJ#3
```

CONTROL PARAMETER

The **control parameter** consists of one of the control break indicators #C1 through #C10. A record field statement containing a control parameter defines a **control field.**

Whenever there is a direct updating (next subsection) of a control field, there is a check for a control break. A **control break** occurs when there is a direct updating of a control field that changes the contents of that field. When such a change takes place, the corresponding control break indicator and all lower-numbered control break indicators are turned on.

However, if the contents of the control field are unchanged by the direct updating, the corresponding control break indicator only is turned off. The lower-numbered control break indicators remain unchanged from their previous states.

The first time that there is a direct updating of a control field, the indicator #F, rather than a control break indicator, is turned on. Any subsequent updating turns indicator #F off and the corresponding control break indicator(s) on.

when a READ CARD statement that references a record with a control field encounters the last data card (which has /* in columns 1 and 2), a control break for that field occurs.

NOTE

Data placed in one of a set of overlapping fields does not constitute a direct updating of the other fields. Such other fields are thus not checked for a control break even though their contents can be changed by the new data.

SEQUENCE PARAMETER

The **sequence parameter** consists of one of the characters >, =, or < followed by an indicator symbol. It specifies that the field is to be checked for sequence during a direct updating (next subsection). The specified indicator is turned on if the new contents of the field are greater than, equal to, or less than, respectively, its previous contents. All fields are initialized to blank (lowest value of the collating sequence) unless explicitly set as a literal field.

For example, the sequence parameter > #50 specifies that indicator #50 is turned on when a direct updating of the field increases the value of its contents. If the direct updating does not change the contents of the field, or decreases them, indicator #50 is turned off.

NOTE

Data placed in one of a set of overlapping fields does not constitute a direct updating of the other fields. Such other fields are thus not checked for sequence even though their contents can be changed by the new data.

LITERAL PARAMETER

The **literal parameter** consists of a literal used to initialize a field. Positions not initialized by the literal are cleared (i.e., contain blanks).

Examples:

```
(5,17),'TOTAL HOURS='
(30,40),'TOTAL COST='
```

Direct Updating of Record Fields

A **direct updating of a record field** occurs whenever:

a. A READ CARD procedural statement (see procedural statement section) causes data to be input to any part of a record containing the field.

b. Data are placed directly in the field by a MOVE, MOVEZ, POST, COLLECT, or ADD procedural statement.

When a direct updating of a field has occurred, the auditing, editing, control-break-checking, and/or sequence-checking specified in the record field statement is performed.

The direct updating is *numeric* when a numeric field or computation is placed directly in a numeric field. Editing, when specified, occurs only in numeric direct updating.

All other types of direct updating, including reading a record, are *alphanumeric*. Auditing, when specified, occurs only in alphanumeric direct updating.

Checking for control breaks and sequencing, when specified, occur in both types of direct updating. Comparisons are algebraic for numeric direct updating and for changes to numeric fields caused by a READ CARD statement being executed. Other comparisons are alphanumeric, following the collating sequence given in appendix B.

NOTE

Data placed in one of a set of overlapping fields does not constitute a direct updating of the other fields. Such other fields are thus not checked even though their contents can be changed by the new data.

TABLE STATEMENT

A **table statement** identifies a table and specifies its size, i.e., the maximum number of entries it will accommodate. The entries in a table all have the same field format as defined by the table field statements (next subsection) that follow the table statement. The format of a table statement is

TABLE name (size) *overflow*

where **name** is the name of the table, **size** is that maximum number of entries in the table, and *overflow* is a general table overflow indicator (# 1 through # 99).

The length of each entry in the table is equal to the minimum space required for all of the table fields as specified by the table field statements.

When *overflow* is used, the general table overflow indicator is turned on when the table is referenced using an index value of less than one or greater than **size**.

Reference to a table entry is made by subscripting the table name. The subscript is a computational expression (section 1) enclosed in parentheses following the table name. The integer portion of the result of the computation specifies the entry referenced, e.g., one for the first entry, four for the fourth entry, etc. When a table name is used without a subscript, the implied subscript is used as a reference. The implied subscript references the last entry in that table found by a LOOKUP statement or input by an ENTER statement (see procedural statement section). For example, TABX(J/2) references the (J/2)th entry in the table TABX, but TABX alone references the last entry looked up or entered.

The effect that the procedural statements ENTER, DELETE, and LOOKUP have on tables is explained under the referenced sections, and depends on the type of table involved. Here are two types of table:

a. A **sequential table** is specified by designating, in a table field statement, one of the fields in each entry of the table as the **key field**. The entries in the table are placed in order according to the ascending values of the key field. Any entry is liable to manipulation.

b. A **last-in-first-out** (**LIFO**) table is specified by *not* designating a key field. The entries in the table are placed in order of their manipulation, i.e., the last entry determines the next so that the changes to the table are at the upper end of the entry sequence. (The current highest entry address is normally set by the last ENTER or DELETE statement affecting this table, but any reference to the table changes the current highest entry address to the value of the subscript in the reference when it is higher than the then-current highest entry address in the table.)

For example, the table statement

```
TABLE TABLEA (100)#17
```

specifies a table of 100 entries of the format indicated by the following table field statements and that indicator #17 is the table overflow indicator. The table statement itself does not specify the type of table. This is done by the presence or absence of a key field in one of the table field statements. If a key field is present in one of the table field statements, it is a sequential table. If no table field statement contains a key field, it is a LIFO (last-in-first-out) table.

TABLE FIELD STATEMENT

All **table field statements** for a given table directly follow the table statement. They define the fields for the entries in the table. The fields can appear in any order and can overlap. However, all entries in a table have the same field format.

The *table field statement for an alphanumeric field* has the format

field (first,last),*post*,*KEY*

where **field** is the name of the field, **first** is the number of the first character position in the field, **last** is the number of the last character position in the field, *post* is the letter P plus a general indicator used for conditional posting like the P parameter of a record field statement and *KEY* specifies that the table is sequential and that this is the *key field* in the table.

KEY in a table field statement indicates the **key field** of a sequential table. The table field statement containing *KEY* is the field used for searching by procedural statements. *KEY*

can be used for only one field per table. If *KEY* is not present in any table field statement for a given table, it is a LIFO (last-in-first-out) table.

The *table field statement for a numeric field* has the format

field (first,last.decimal),*post*,*KEY*

where the definitions are as above except that **last** is followed by a decimal point and **decimal**, which specifies the number of digits to the right of the implied decimal point in the field. It is present for every numeric field, even if the value is zero. If the field contains an actual decimal point, its position overrides the specification in the table field statement.

An example of a table field statement for an alphanumeric field is

```
ACCNT (1,10),KEY
```

which specifies that the field ACCNT occupies character positions one through ten in each entry in the table, and that ACCNT is the key field for this table.

An example of a table field statement for a numeric field is

```
AMOUNT (11,17.2),P#32
```

which specifies that the field AMOUNT occupies character positions 11 through 17 in each entry in the table, and that there are two digits to the right of the implied decimal point in the field. Indicator #32 is the conditional posting indicator. This field is not a key field.

PROCEDURAL STATEMENTS

Procedural statements follow the data-defining statements. They direct the execution of the program as it processes the data previously defined by the data-defining statements.

The **PROCEDURE** statement is the first procedural statement and comprises the single word PROCEDURE. It serves as the divider between the data-defining statements and the procedural statements.

Subsequent procedural statements manipulate the data to obtain the desired output. They have the general form

statement number (condition) **VERB direction**

where the optional *condition* specifies the condition(s) under which the statement is to be executed, **VERB** specifies the action to be taken, and **direction** specifies the object(s) of the verb. If no condition is specified, the statement is executed unconditionally. The formats, elements, and uses of individual procedural statements are explained in the

following subsections. Any procedural statement can begin with an optional statement number.

The **END** statement is the last procedural statement. It indicates the last input to the RPG IV language processor.

If the verb of a procedural statement is to be repeated in subsequent statements, it can be replaced by the ditto mark (").

Example:

```
(#3 OR #44) MOVE C,D
(#3 OR #44) ' A, B
```

The ditto mark cannot be used for repetition of directions.

The #" indicator can be used for repetition of a condition.

Example:

```
(#3 OR #44) MOVE C,D
(#')' A, B
(#') COMPUTE X = J+TOTAL
```

PROCEDURE STATEMENT

This statement is always the first procedural statement. Thus it directly follows the last data-defining statement and serves as a divider between the two types of statement. PROCEDURE terminates the processing of data definitions and begins the processing of procedural manipulations. This statement has only one form:

```
PROCEDURE
```

MOVE STATEMENT

This statement moves a literal, a constant, or the contents of one field to another field or fields. It has the format

(condition) **MOVE from,to,***to,...*

where **from** is the literal or constant to be moved, or the name of a field whose contents are to be moved; and **to** is the name of the field to receive the moved item. The movement can be made to additional fields by entering the names of all such fields, separated by commas, in the MOVE statement after the first **to**.

A **numeric movement** occurs when the from-field contains a constant or the name of a numeric field, and the to-field is a numeric or implied field. A numeric movement moves only numeric information, re-editing and rescaling it to the format of the to-field.

An **alphanumeric movement** occurs in all other cases. However, constants or implied numeric fields cannot be moved to explicit alphanumeric fields. An alphanumeric movement moves the characters one by one, left to right. Movement stops when the to-field is full. If the from-field is shorter than the to-field, the to-field is filled out with blanks after all characters have been moved from the from-field.

Examples:

```
            MOVE 'TITLE',FIELD1,FIELD2
(TIME=10) MOVE IN.JOB,OUT.JOB
            MOVE INVAL,OUTVAL
```

MOVEZ STATEMENT

This statement moves *only the zone bits* of the characters in a literal or field to the zone bits of corresponding characters in another field or fields. It has the format

(condition) **MOVEZ from,to,***to,...*

where **from** is the literal or field whose zone bits are to be moved, and **to** is the name of the field to receive the moved zone bits on corresponding characters. The movement can be made to additional fields by entering the names of all such fields, separated by commas, in the MOVEZ statement after the first **to**.

The MOVEZ statement does not apply to numeric movements. It operates as an alphanumeric movement under a MOVE statement except that only the zone bits of the characters are moved. (Zone bits are those corresponding to the 11- and 12-punches on punched card input.)

Examples:

```
(#1 OR X=5) MOVEZ 'A',XYZ
                  MOVEZ NEWSUM,CODE1,CODE4
```

POST STATEMENT

This statement posts the contents of fields in one record or table entry to *like-named* fields in other records or entries. It has the format

(condition) **POST from,to,***to,...*

where **from** is the name of the record or entry from which the posting is made, and **to** is the name of a record or entry to which the posting is made. The posting can be made to

additional records or entries by entering the names of all such records or entries, separated by commas, in the POST statement after the first **to**. Posting obliterates the original contents of the to-fields, and replaces them with the contents of the from-field.

The field moves as under a MOVE statement except when the field in the to-record or to-entry has a P parameter that is off.

Examples:

```
(#C3)     POST INPUT, OUTPUT
(#")      POST INPUT, OUTPUT,TOTAL
          POST BOOKS(2*X),CURRENT
```

COLLECT STATEMENT

This statement adds the contents of fields in one record or table entry to the contents of *like-named* fields in other records or entries. It has the format

(condition) **COLLECT from,to,***to...*

where **from** is the name of the record or entry whose contents are to be added, and **to** is the name of a record or entry whose contents are to be augmented by the amount contained in **from**. The addition can be made to additional records or entries by entering the names of all such records or entries, separated by commas, in the COLLECT statement after the first **to**. The COLLECT statement functions like the POST statement except that the to-fields after execution contain the sum of their former contents and the contents of the from-field.

The field moves as under a MOVE statement except when the field in the to-record or to-entry has a P parameter that is off. If any significant digits of the result are lost because the to-field is not large enough to hold them, computational overflow indicator #X1 comes on. If either field is a nonnumeric implied field, mode error indicator #X2 comes on.

Examples:

```
(#17)     COLLECT DETAIL, MASTER
          COLLECT ACCT,DEPT,TOTAL
```

ADD STATEMENT

This statement adds a constant or the contents of one field to the contents of other fields. It has the format

(condition) **ADD from,to,***to,...*

where **from** is the constant or the name of the field whose contents are to be added, and **to** is the name of the field containing the value to which the addition is made. The

addition of **from** can be made to the contents of several fields by entering the names of all such fields, separated by commas, in the ADD statement after the first **to**.

The accuracy of the result depends on the size of the to-field and the position of the decimal point in it. If any significant digits of the result are lost because the to-field is not large enough to hold them, computational overflow indicator #X1 comes on. If either field is a nonnumeric implied field, mode error indicator #X2 comes on.

The ADD statement can be coded as a COMPUTE statement for updating a field as follows:

```
COMPUTE to = to+from
```

Examples:

```
          ADD MONTH,YTD
(#C1)     ADD ACCT.AMT,DEPT.AMT
          COMPUTE ZAP =ZAP+1
```

COMPUTE STATEMENT

This statement computes the value of an expression and places the result in the specified field. It has the format

(condition) **COMPUTE field = expression**

where **field** is the name of the numeric or implied field receiving the result of the computation, and **expression** is the computational expression, constant, or numeric field being evaluated.

The result is moved as for a numeric move under a MOVE statement. Editing occurs if it has been specified for the specified field.

When the result is plus, zero, or minus, the corresponding indicator (#P, #Z, or #M) is turned on and the other two turned off.

Computational overflow turns the #X1 indicator on.

Examples:

```
(#C3)          COMPUTE AH = AH+EH
(#2 OR #5)     COMPUTE A(I) = 10*B+(2*C)
               COMPUTE DETAIL.COST = RATE*HOURS
               COMPUTE TOTAL = X(1)+X(2)
```

GO TO STATEMENT

This statement alters the flow of the program by specifying the next statement to be executed. It has the format

(condition) **GO TO number**

where **number** is the number of the next statement to be executed.

Examples:

```
(NOT #E)        GO TO 1000
(A=B)           ' 777
                GO TO 255
```

INDEXED GO TO STATEMENT

This statement, like the GO TO statement, alters the flow of the program by specifying the next statement to be executed. It has the format

(condition) **GO TO (number,***number,...***)index**

where **number** is the number of a statement that can be the statement selected by the value of the **index**, which is the name of a numeric field. More than one statement can be included by entering additional statement numbers, separated by commas, after the first **statement**, but within the parentheses.

The integer portion of the contents of **index** selects the number of the statement to be executed next. Thus, if this value is one, two, three, etc., the first, second, third, etc., statement number, respectively, is the number of the statement to be executed next. For instance, in the first example below, the next statement executed is statement number 20 if the value of the integer portion of the contents of X is two. If the index is out of range, the program continues in sequence and next executes the statement following the indexed GO TO statement.

Examples:

```
                    GO TO (10,20,30)X
(#3 OR NOT #7)      " (101,35,972,15)INPUT.KEY
(#')                "   (11,22,33,500)UU78
```

PERFORM STATEMENT

This statement, like the GO TO statement, alters the flow of the program by specifying the next statement to be executed. In addition, however, PERFORM stores the present address of the program so that when a RETURN statement (next subsection) is found in the

sequence following the specified statement, the program flow returns to the main sequence at the statement following the PERFORM statement. The format of the PERFORM statement is

(condition) **PERFORM number**

where **number** is the number of the next statement to be executed. Note that this statement is always used in conjunction with a RETURN statement. If the RETURN statement is missing, the effect of the perform statement is that of a GO TO statement.

Examples:

```
(A >< B)        PERFORM 95
                PERFORM 7250
```

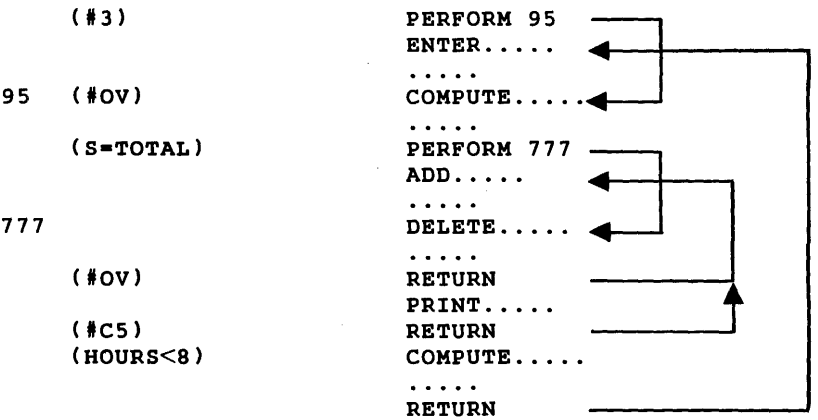
RETURN STATEMENT

This statement returns the flow of the program to the statement following the corresponding PERFORM statement. The RETURN statement has the format

(condition) **RETURN**

There is at least one RETURN statement for each PERFORM statement, but only one of these RETURN statements is executed on each pass. The PERFORM statement places the return location value in a LIFO (last-in-first-out) queue. The following RETURN statement uses the last such location placed in the queue. Thus, subroutines can be nested using these two statements.

Example (arrows show the flow of the program when the condition of the statement is met):



SET STATEMENT

This statement turns indicators on or off under specified conditions. It has the format

(condition) **SET value indicator,***indicator,...*

where **value** is the word ON, the word OFF, or a conditional expression within parentheses; and **indicator** is the symbol for an indicator. More than one indicator can be specified by entering their symbols, separated by commas, after the first **indicator**.

If **value** is ON and the statement *condition* is met, the specified indicators are turned on. If **value** is OFF and the statement *condition* is met, the specified indicators are turned off. If **value** is a conditional expression that is true and the statement *condition* is met, the specified indicators are turned ON, but if the condition expression is not true, the specified indicators are turned OFF. In any case, if the statement *condition* is not met, the indicators are unchanged because this statement will not be executed.

If a control break indicator is turned on or off by a SET statement, there is no change in the status of any other control break indicators, i.e., the lower-level control break indicators are unaffected by a SET statement unless they are explicitly specified therein.

Examples:

```
(#C1) SET ON #1, #2, #58
        SET (HR<8.0 AND T>500) #26
(#C6) SET OFF #LC
```

ENTER STATEMENT

This statement assigns space for a new entry to a table and posts data into the new entry. It has the format

(condition) **ENTER record,table,***index*

where **record** is the name of the table entry or record from which the posting is made, **table** is the name of the table into which the new entry is being posted, and *index* is the name of a numeric or implied field that is set equal to the number (subscript) of the new entry.

The address of the new entry depends on the type of table (see table statement description):

a. In *a LIFO (last-in-first-out) table*, the address is one greater than that of the current highest entry or subscript reference value.

(continued)

b. In *a sequential table*, the **record** contains a field having the same name as that of the table's key field. The value of this field determines where the new entry is assigned. When this value matches an existing key in the table, the new entry overlays that position. Otherwise, all entries having higher key values move up one position to make room for the new entry.

After the address of the new entry is established, the ENTER statement posts the contents of **record** to that entry just as under a POST statement. Data can also be entered in a table by subscript referencing without the use of ENTER statements.

If the new entry would cause the table to overflow, no posting occurs and the table overflow indicator (a general indicator you will have defined) is turned on. This indicator can be turned off only by a SET OFF statement.

After a successful entry, the implied subscript for the table references the new entry.

Examples:

```
            ENTER INPUT,TABLEA
(#22)       ENTER DATA,CTROL,IX55
```

DELETE STATEMENT

This statement deletes an entry from a table. It has the format

(condition) **DELETE table***(subscript)*,*key*

where **table** is the name of the table from which the deletion is to be made; *subscript* (enclosed in parentheses) is the number of the entry to be deleted; and *key* is a field name, constant, or literal used to find the entry to be deleted. A *key* is used only for sequential tables in the absence of a *subscript*.

When there is neither *subscript* nor *key*, the entry to be deleted is specified by the implied subscript, and depends on the type of table:

a. In *a LIFO table*, the entry deleted is the current highest entry. Deletion reduces the current highest entry by one.

b. In *a sequential table*, the entry deleted is the last entry entered or looked up.

In any case, all entries above the deleted entry move down one position in the table after the entry is deleted.

Examples:

```
                DELETE TABLEA
(#42 AND #43)   DELETE CTROL,MONTH
```

LOOKUP STATEMENT

This statement determines, in a sequential table, if there is an entry having a key field (section 1.4) equal to or greater than the key specified in the LOOKUP statement, and sets indicators according to the findings. The format of the LOOKUP statement is

(condition) **LOOKUP table,key,***index*

where **table** is the name of the sequential table to be searched; **key** is a field name, constant, or literal compared with the values in the table entry key fields; and *index* is the name of an implied or numeric field that is set equal to the implied subscript found (see below).

The implied subscript for the table and the (optional) index are set to reference the first entry having a **key** equal to or greater than that of the one specified in the LOOKUP statement. If there is no such entry in the table, the implied subscript and index are set to reference the last entry address.

The LOOKUP statement sets the #E, #G, and #L indicators as follows:

a. If a match is found between the specified key and that of a table entry, the #E (equal) indicator is turned on and the other two are turned off.

b. If no match is found but there is a table entry having a key field greater than the specified key, the #G (greater than) indicator is turned on and the other two turned off.

c. In other cases, the #L (less than) indicator is turned on and the other two turned off. This is always the case for empty tables. For full tables, the implied subscript is set to the maximum table entry plus one.

The LOOKUP statement is not applicable to LIFO tables.

Examples:

```
          LOOKUP TABLEA,INPUT.DEPT
(#E)      LOOKUP CTROL,MONTH.IND
```

CALL STATEMENT

This statement calls a DAS-coded subroutine (appendix E). It has the format

(condition) **CALL subroutine,***argument,...*

Where **subroutine** is the name of the DAS-coded subroutine being called, and argument is a constant, name of a record or table, or a nonsubscripted field. More than one argument can be included by entering additional arguments, separated by commas, after the first.

The subroutine must be written in DAS assembler language and provides a method of augmenting the RPG language with special functions

Subroutines referenced by the RPG IV CALL statement must be manually included with the runtime package prior to loading. Appendix E describes the detailed procedures.

Examples:

```
(#3)   CALL FACTOR,PARM,10.3
(#')   CALL PACK
         CALL EXIT
(#OV)  CALL SQRT,INPVAL,OUTVAL
(#')   CALL CLOCK,TIME
         ' SQRT,XXX
```

READ CARD STATEMENT

This statement reads a card and inputs the data on the card to each of the specified records whose acceptance criteria are met. It has the format

(condition) **READ CARD name**,*name*,...

where **name** is the name of the record to receive the data provided its acceptance criteria are satisfied. The data can be read into more than one record by entering additional record names, separated by commas, after the first **name**.

After the card is read, each record specified accepts or rejects the data on the basis of its own selection criteria. If the data are accepted by a record, it enters the record without editing and truncates the right end of the card image if the record is too small to hold all the data.

Control break checks, sequence checks, and auditing checks are performed as required by the record field statements.

If the card read is the last data card (columns 1 and 2 contain /*), the card image is not read into the records. The #LC indicator and all control break indicators associated with the specified records turn on.

Examples:

```
(#25)     READ CARD EMPLYE
(#")      READ CARD MAN,WOMAN,CHILD
```

Under MOS, the READ CARD statement causes an alphanumeric read operation from logical unit 16. If unit 16 is assigned to a card reader, up to 80 characters can be input. If unit 16 is not assigned to a card reader, the record length is limited by the device or the

length defined by the RECORD (plus field) statements, whichever is less. In the stand-alone version, the limit is 80 characters.

RPG IV data read by stand-alone and MOS versions are input and converted as EBCDIC (029 card) codes.

Under MOS, if a record is read with a (0-1) punch in column one, the program is aborted and control returned to MOS. The only exception is if the record contains a /* in columns 1 and 2, then the #LC indicator is set.

Under VORTEX, the READ CARD statement causes an ASCII read operation from logical unit 13. If unit 13 is assigned to a card reader, up to 80 characters can be input. If unit 13 is not assigned to a card reader, the record length is limited by the device or the length defined by the RECORD (plus field) statements, whichever is less. For READ CARD logical unit 13 cannot be a rotating memory device.

RPG IV data read by the VORTEX version will be converted as BCD or EBCDIC (026 or 029) card codes depending upon the mode selected by the /KPMODE directive.

PRINT STATEMENT

This statement performs page control functions or prints data or messages on the line printer. It has the format

(condition) **PRINT parameter**,*parameter*,...

where **parameter** is one of the following and the line printer performs the function indicated for the parameter:

Record Name The record is printed, the paper advanced one line, and record fields having a B(blank after print) parameter are cleared. If, by advancing the paper, a line count of 44 is reached, the #OV indicator is set on. Under MOS, and VORTEX the line count can be altered by the /FORM directive allowing the #OV indicator to be set on any line count.

$An (n = 1-7) The paper advances n lines. If the bottom of the page is reached by the advancement of the paper, the #OV indicator comes on. If n is omitted or is zero, the paper advances one line. Value greater than 7 will cause a compilation error.

$Cn (n = 1-7) The paper advances to the designated line position on the page as determined by the vertical-format tape in the line printer. If N = 1, the paper advances to the top of the next page and the #OV indicator

(continued)

goes off. If N = 7, the paper advances to the position determined by channel 7 of the tape and the #OV indicator comes on. Under VORTEX, the vertical format tape channel used to control the slew is one less than N, i.e., if N = 1, the paper is advanced to a point determined by channel 0 on the tape.

Additional parameters can be specified in the same PRINT statement by entering the parameters, separated by commas, after the first **parameter**.

Examples:

```
(#OV)     PRINT $C1,HEADER,$A1
          PRINT DETAIL
```

Under MOS, the PRINT statement causes an alphanumeric write operation on logical 18. The record length is limited by the device or the length defined by the RECORD (plus field) statements, whichever is less. In the stand-alone version, the record limit is 132 characters.

Under VORTEX, the PRINT statement causes an ASCII write operation to logical unit 15. The record length is limited by the device or the length defined by the record (and field) statements, whichever is less. A leading space character is appended to the print line since the system uses the first characters as a format control, thus the output to the printer is shifted one column to the right. Output with PRINT statement is limited to 132 characters, and cannot have logical unit 15 assigned to a rotating memory device.

PUNCH STATEMENT

This statement punches one or more cards. It has the format

(condition) **PUNCH name**,*name*,...

Where *name* is the name of the record to be punched. More than one card can be punched with a single PUNCH statement by entering additional record names, separated by commas, after the first **name**.

Examples:

```
(KEY=3)   PUNCH SUMMARY
          PUNCH DATA, SUMMRY,H44
```

Under MOS, the PUNCH statement causes an alphanumeric write operation on logical unit 17. The record length is limited by the device or the length defined by the RECORD (plus field) statements, whichever is less. In the stand-alone versions, the record limit is 80 characters.

RPG IV data punched by stand-alone and MOS versions are converted and output as EBCDIC (029) card codes.

Under VORTEX, the PUNCH statement causes an ASCII write operation to logical unit 14. The record length is limited by the device or the length defined by the record (plus field) statements, whichever is less. For RPG PUNCH cannot have logical unit 14 as a rotating memory device.

RPG IV data punched by the VORTEX version will be converted as BCD or EBCDIC (026 or 029 keypunch) card codes depending upon the mode selected by the /KPMODE directive.

STOP STATEMENT

This statement stops the execution of the program and outputs a message to the computer operator. If the operator presses RUN after a STOP, the program continues execution with the statement following the STOP. The STOP statement has the format

(condition) **STOP** *message*

where *message* is the alphanumeric field or literal output to the computer operator.

Examples:

```
          STOP 'END OF RUN'
(#C1)     STOP 'OUT OF DATA'
(#LC)     STOP
```

In the stand-alone version, messages are output to the line printer. Under MOS, they are directed to the list output (LO) device.

Under VORTEX, the runtime execution of the STOP statement causes the program to execute a SUSPND call. The program may be continued by use of the RESUME command in OPCOM, i.e., ;RESUME, RPGRT. The STOP message is output to the LO device.

END STATEMENT

This statement is always the last procedural statement, and, therefore, the last statement in the RPG IV program. It is not executable and serves only to indicate the end of the program. This statement has only one form:

```
END
```

MOS I/O CALLS

The stand-alone version of RPG IV provides I/O statements for reading, punching, and printing. Used under MOS, these statements allow one input file and two output files.

The MOS version of RPG IV expands this I/O capability by providing six CALL statements for performing additional I/O operations. Through the use of these calls and proper device assignments (see MOS Manaual, /ASSIGN directive), the user can manipulate up to ten files at one time, any of which can be on such devices as magnetic tape units and disc.

The following subsections provide detailed descriptions of each CALL statement. All parameters must be supplied and must be the proper type. Errors in CALL statements are indicated by the message:

```
INVALID RPG CALL TO xxxxxx
```

where xxxxxx specifies the name of the called subroutine as it appears in the CALL statement. The error message is logged on LO and the job is aborted. Figure 3-1 is a sample RPG IV program that utilizes MOS I/O calls.

Read Alphanumeric Record

```
CALL READ,lu,record,size,exception
```

where:

lu	=	logical unit number (16-25)
record	=	name of record as it appears in a RECORD statement
size	=	number of characters in record (must be in even number, if not, the last character is truncated).
exception	=	name of implied field set after read operation:
		0 = reading successful
		1 = irrecoverable reading error
		2 = end of file detected
		3 = end of device detected

Note: If the first character read is a (/), 0-1 punch and it is not followed by an *, the program is aborted and control returned to MOS.

Write Alphanumeric Record

```
CALL WRITE,lu,record,size,exception
```

where:

lu	=	logical unit number (16-25)
record	=	name of record as it appears in a RECORD statement
size	=	number of characters in record (must be in even number, if not, the last character is truncated).
exception	=	name of impled field set after write operation: 0 = writing successful 1 = irrecoverable writing error 3 = end of device detected

Write End of File

```
CALL WEOF,lu,exception
```

where:

lu	=	logical unit number (16-25)
exception	=	name of implied field set after end of file is written: 1 = irrecoverable writing error 3 = successful execution of WEOF

Rewind Unit

```
CALL REWIND,lu,exception
```

Where:

lu	=	logical unit number (16-25)
exception	=	name of implied field set after rewind: 1 = irrecoverable error 3 = successful rewind

Skip Record

```
CALL SKIPR,lu,count,direction,exception
```

Where:

lu	=	logical unit number (16-25)
count	=	number of records to be skipped
direction	=	0 for forward, not 0 for backward
exception	=	name of implied field set after record(s) skip: 0 = successful execution 1 = irrecoverable error 2 = end of file detected 3 = end/beginning of device detected

Skip File

```
CALL SKIPF,lu,count,direction,exception
```

Where:

lu	=	logical unit number (16-25)
count	=	number of files to be skipped
direction	=	0 for forward, not 0 for backward
exception	=	name of implied field after file(s) skip: 0 = successful execution 1 = irrecoverable error 3 = end/beginning of device detected

EXIT TO MOS

An RPG IV program run under MOS is terminated by a special call. It has the format:

```
CALL EXIT
```

and returns control to the MOS executive.

```
*          RPG IV PROGRAM FOR COMPARING TWO FILES ON LOGICAL UNITS
*          20 AND 21, AND PRINTING MISMATCHES ON 18.  ERRORS ARE LOGGED
*          ON LOGICAL UNIT (LO) AND INPUT IS TERMINATED BY AN EOF.
*          RECORDS ARE 100 BYTES LONG.
*
           RECORD FILE1
             FIELD1 (1,100)
           RECORD FILE2
             FIELD2 (1,100)
           RECORD HEAD
             (1,16) 'MISMATCHES ON 21'
           RECORD MSG1
             (1,19) 'EOF ON 20 BEFORE 21'
           RECORD MSG2
             (1,19) 'EOF ON 21 BEFORE 20'
           RECORD MSG3
             (1,10) 'END-OF-JOB'
*
           PROCEDURE
*
*          REWIND INPUT FILES
*
1          CALL REWIND,20,EX
(EX=1)  STOP 'REW ER-20'
(#')                    GO TO 1
2          CALL REWIND,21,EX
(EX=1)  STOP 'REW ER-21'
(#')                    GO TO 2
*
*          READ INPUT RECORDS
*
3          CALL READ,20,FILE1,100,EX
(EX=1 OR EX=3)          STOP 'READ ER-20'
(#')                    GO TO 3
(EX=2)  SET ON #1
4          CALL READ,21,FILE2,100,EX
(EX=1 OR EX=3)             STOP 'READ ER-21'
(#')                    GO TO 4
*
*          TEST FOR EOF CONDITIONS
*
(#1 AND EX > < 2)  PRINT $C1,MSG1,$C1
(#')                    CALL EXIT
(NOT #1 AND EX=2)     PRINT $C1,MSG2,$C1
(#')                    CALL EXIT
(#1 AND EX=2)         PRINT $C1,MSG3,$C1
(#')                    CALL REWIND,20,EX
(#')                    CALL REWIND,21,EX
(#')                    CALL EXIT
*
*          COMPARE RECORDS AND PRINT MISMATCHES
*
(FIELD1 = FIELD2)       GO TO 3
(#OV)   PRINT $C1,HEAD,$A1
          PRINT FILE2
          GO TO 3
          END
```

Figure 3-1. RPG IV Program Utilizing MOS I/O Calls

VORTEX RPG IV

In addition to the READ CARD, PUNCH and PRINT statements, the VORTEX version of RPG expands this I/O capability by providing seven CALL statements for performing additional I/O operations.

These additional I/O operations are performed on logical units 16 through 22. The following paragraphs provide a detailed description of the CALL statements. All parameters must be supplied and must be of the proper type. An error in the use of a CALL statement will result in an error message (see below) being posted on logical unit number 15 and the job being terminated.

The seven I/O CALL statements are:

CALL OPEN, u, lun, filename, key, record size, access method, mode, exception

where:

u	=	RPG unit number (16-22)
lun	=	VORTEX logical unit number
filename	=	a name of a field containing a six-character literal string which is the filename.
key	=	a name of a field containing a one-character literal string which is the protection key for the file.
record size	=	the number of characters in the record (must be an even number, if not, the last character is truncated).
access method	=	the manner in which the file is to be accessed; 0 if direct access by logical record, 1 if sequential access by logical record.
mode	=	the mode in which the OPEN operation is to be performed; 0 if open and rewind, 1 if open and leave.

(continued)

exception = name of an implied field set to the following values at the completion of the open request 0 = open successful, 1 = irrecoverable error.

CALL CLOSE, u, mode, exception

where:

u = RPG unit number (16-22)

mode = the mode in which the CLOSE operation is to be performed 0 if close and leave, 1 if close and update.

exception = name of an implied field set to the following values at the completion of the close request 0 = close successful, 1 = irrecoverable error.

CALL READ/WRITE, u, record name, record size, record number, exception

where:

u = RPG unit number (16-22)

record name = name of record as it appears in RECORD statement.

record size = number of characters in record (must be an even number, if not, the last character is truncated).

record number = the name of an implied field which contains the record number to be read or written in the direct access mode; if the file is being accessed in the sequential mode, the record number should be zero.

(continued)

exception = name of an implied field set to the following values at the completion of the READ or WRITE request;
0 = READ/WRITE successful
1 = irrecoverable error
2 = End of file detected
3 = End of device detected

CALL WEOF, u, exception

where:

u = RPG unit number (16-22)

exception = name of an implied field set to the following values at the completion of the WEOF request; 0 = WEOF successful, 1 = irrecoverable error.

CALL REWIND, u, exception

where:

u = RPG unit number (16-22)

exception = name of an implied field set to the following values at the completion of the REWIND request; 0 = rewind successful, 1 = irrecoverable error.

CALL SKIPR, u, record count, direction, exception

where:

u = RPG unit number (16-22)

record count = number of records to be skipped

direction = 0 if forward; non-zero if backward

exception = name of an implied field set to the following values at the completion of the SKIP RECORD request;
0 = SKIP RECORD successful
1 = irrecoverable error
2 = End of file detected
3 = End of device detected

The OPEN call links the RPG unit number with the VORTEX logical unit number (so that multiple files within an RMD partition may be accessed).

The CLOSE call unlinks the RPG unit number and the VORTEX logical unit number.

If a prior OPEN does not bind the RPG unit with a VORTEX logical unit number, it is assumed on subsequent READ, WRITE, etc. that the RPG unit number equals the VORTEX logical unit number.

Input/output initiated by calls do not invoke any of the automatic record checking features of RPG, i.e., control break, audit, etc.

The VORTEX versions of RPG also has the CALL EXIT subroutine to return control to the executive. RPG programs, upon completion, must return control to the executive through the CALL EXIT statement.

Figure 3-2 is a sample RPG IV program that utilizes VORTEX IOC calls.

```
*
*   RPG IV PROGRAM FOR COMPARING TWO FILES ON RPG UNITS
*   20 AND 21, AND PRINTING MISMATCHES ON UNIT 15. ERRORS ARE
*   LOGGED ON (LO) AND INPUT IS TERMINATED BY AN EOF.
*   RECORDS ARE 100 BYTES LONG.
*
    RECORD FILEN1
      NAME1 (1,6) 'FILE1'
      KEY1 (7,7) ' '
    RECORD FILEN2
      NAME2 (1,6) 'FILE2'
      KEY2 (7,7) ' '
    RECORD FILE1
      FIELD1 (1,100)
    RECORD FILE2
      FIELD2 (1,100)
    RECORD HEAD
      (1,16) 'MISMATCHES ON 21'
    RECORD MSG1
      (1,19) 'EOF ON 20 BEFORE 21'
    RECORD MSG2
      (1,19) 'EOF ON 21 BEFORE 20'
    RECORD MSG3
      (1,10) 'END OF JOB'
*
    PROCEDURE
    MOVE 0, REC
```

Figure 3-2. RPG IV Program Using VORTEX I/O Calls

```
*
*   OPEN/REWIND FILES
*
1   CALL OPEN,20,180,NAME1,KEY1,100,1,0,EXCP
    (EXCP=1) STOP 'OPEN ERROR ON 20'
    (#') GO TO 5
2   CALL OPEN,21,180,NAME2,KEY2,100,1,0,EXCP
    (EXCP=1) STOP 'OPEN ERROR ON 21'
    (#') GO TO 5
*
*   READ INPUT RECORD
*
3   CALL READ,20,FILE1,100,REC,EXCP
    (EXCP=1 OR EXCP=3) STOP 'READ ERROR ON 20'
    (#') GO TO 5
    (EXCP=2) SET ON #1
4   CALL READ,21,FILE2,100,REC,EXCP
    (EXCP=1 OR EXCP=3) STOP 'READ ERROR ON 21'
    (#') GO TO 5
*
*
*   TEST FOR EOF CONDITIONS
*
    (#1 AND NOT EXCEP=2) PRINT $C1,MSG1,$C1
    (#') CALL EXIT
    (NOT #1 AND EXCP=2) PRINT $C1,MSG2,$C1
    (#') CALL EXIT
    (1 AND EXCP=2) PRINT $C1,MSG3,$C1
    (#') GO TO 6
*
*   COMPARE RECORDS AND PRINT MISMATCHES
*
    (FIELD1=FIELD2) GO TO 3
    (#OV) PRINT $C1,HEAD,$A1
    PRINT FILE2
    GO TO 3
*
*   CLOSE/UPDATE FILE
*
6   CALL CLOSE,20,1,EXCP
    (EXCP=1) STOP 'CLOSE ERROR ON 20'
    CALL CLOSE,21,1,EXCP
    (EXCP=1) STOP 'CLOSE ERROR ON 21'
5   CALL EXIT
    END
```

Figure 3-2. RPG IV Program Using VORTEX I/O Calls *(continued)*

SECTION 4 - OPERATING PROCEDURES

This section explains how to run the RPG IV programs written according to the instructions in section 3. The program is first compiled and the resulting object deck then used to process the data.

NOTE

In this section, numbers beginning with a zero are octal and numbers beginning with any other digit are decimal. References to START and RESET refer to the V73, 620/f, and 620/f-100 computers. References to RUN and SYSTEM RESET refer to the 620/L and 620/L-100 computers. Refer to the applicable system reference manual for descriptions of the control panel switches and indicators.

COMPILING AN RPG IV PROGRAM

The RPG IV compiler is available in three versions: stand-alone, MOS and VORTEX. The stand-alone version is supplied on cards as a two-part compiler; part I is for data-defining statements and part II is for procedure statements. The MOS version is an integral part of the standard MOS Installation System Library. The VORTEX RPG compiler may be added to the background library (BL) after the system is generated.

The RPG IV compiler is a one-pass compiler that reads a source module (program), produces an object module (executable code), and generates a source listing. The listing includes diagnostic and error messages.

DECK PREPARATION

Deck Preparation for Compilation (Stand-Alone)

The card deck for compilation is shown in figure 4-1 and comprises, in order:

a. Binary card loader (three cards, supplied)

b. RPG IV COMPILER PART I

c. The data-defining statements of the RPG IV program (continued)

d. The PROCEDURE statement of the RPG IV program

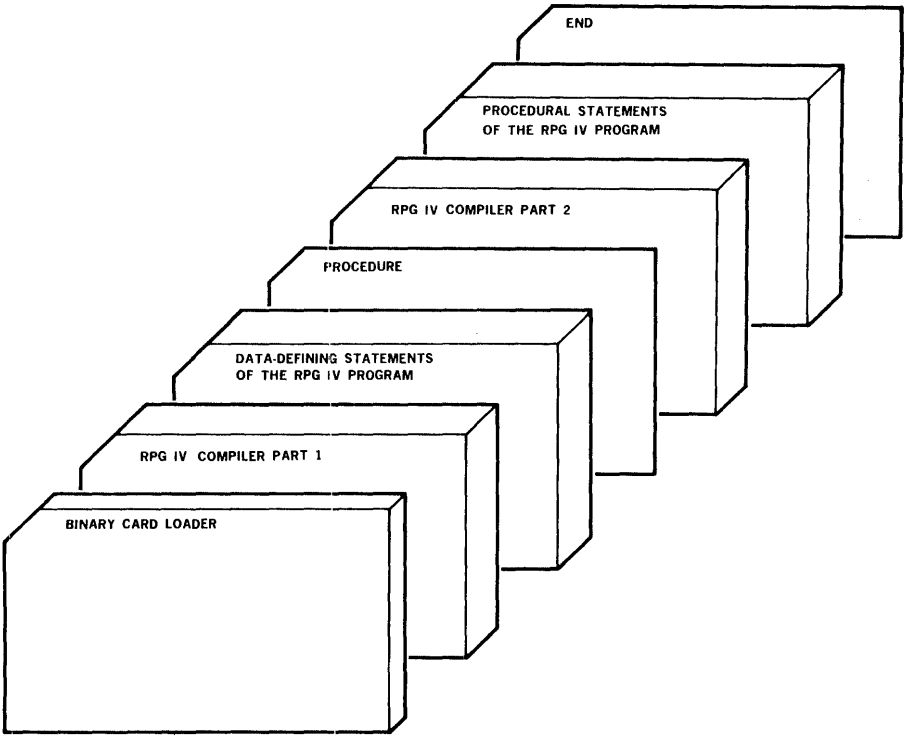
e. RPG IV COMPILER PART II

f. The procedural statements of the RPG IV program

g. The END statement of the RPG IV program

The binary card loader is loaded by the card bootstrap loader (appendix D). The binary card loader then loads the RPG IV COMPILER PART I, which then processes the data-defining statements of the program. When this is completed, the binary card loader loads the RPG IV COMPILER PART II, which then processes the procedural statements of the program.

These processes yield a printed listing of the program and an object deck of the compiled program.



VTII-1010

Figure 4-1. Deck for Compiling an RPG IV Program (Stand-Alone Version)

Deck Preparation for Compilation (MOS Version)

The MOS RPG IV compiler reads source records from the Processor Input file (PI), writes object records on the Binary Output file (BO), and lists the source program on the List Output file (LO). These logical units can be assigned to any valid MOS peripheral. By assigning it to dummy, a logical unit can be disabled. For example, /ASSIGN,BO = DUM would result in compilation with the BO suppressed.

The compiler input is terminated by an END statement and control returns to MOS. Therefore, it is necessary to reload the compiler with another /ULOAD,RPGC directive if multiple compilations are desired.

A sample job stream for an MOS compilation is shown in figure 4-2 and comprises, in order:

a. An optional job card to identify jobs in the input stream.

b. An optional date card (the date is printed at the top of the source program listing if one is supplied).

c. An optional forms card if other than the default value for number of lines per page is desired.

d. An optional assignment card if assignments other than the default peripheral assignments are required, or if assurance of additional assignments is desired.

e. An unconditional load card to direct the loading of the RPG IV compiler from the system file.

f. The RPG IV source program.

g. An optional end-of-job card to separate jobs in the input stream.

Deck Preparation for Compilation (VORTEX Version)

The VORTEX RPG IV compiler and the VORTEX RPG IV runtime/loader execute as level 0 background programs in unprotected memory. Both the compiler and runtime/loader will operate in 6K of memory with limited work stack space. The stack space may be expanded and consequently larger RPG programs compiled and executed, by use of the VORTEX /MEM directive.

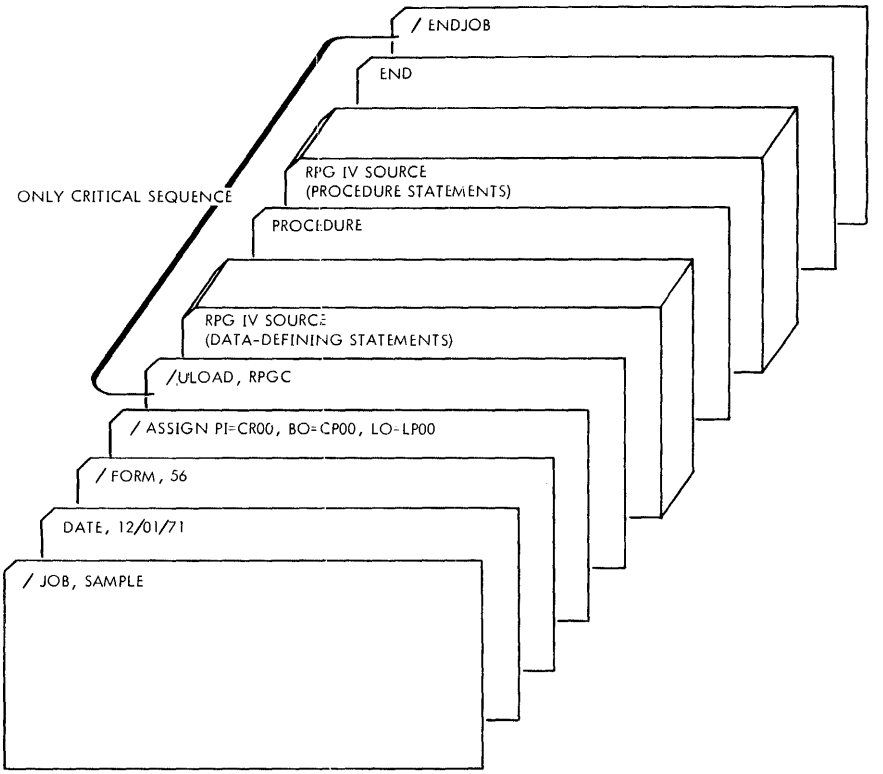
The VORTEX RPG IV compiler reads source records from the Processor Input device (PI), writes object records on the Binary Output device (BO), and lists the source program, and any diagnostic or error messages, on the list output device (LO). If PI is assigned to a Rotating Memory Device (RMD) partition, the compiler assumes the source records are blocked three 40-word records per RMD 120-word physical record. However, if PI is the

same logical unit as the system input device (SI), and is assigned to a Rotating Memory Device (RMD) partition, the compiler assumes the source records are not blocked but consist of one source record per RMD 120-word physical record. If BO and/or LO is assigned to a RMD partition, the output is blocked two 60-word records per RMD 120-word physical record.

If PI is assigned to a card reader during compilation, the /KPMODE directive maybe used to indicate whether 026 or 029 keypunched cards are to be read.

The compiler is scheduled from the background library at level by the /LOAD, RPGC directive. The compiler terminates when an END statement is encountered, and exits to the executive. Only one RPG IV program can be compiled for each load of the compiler.

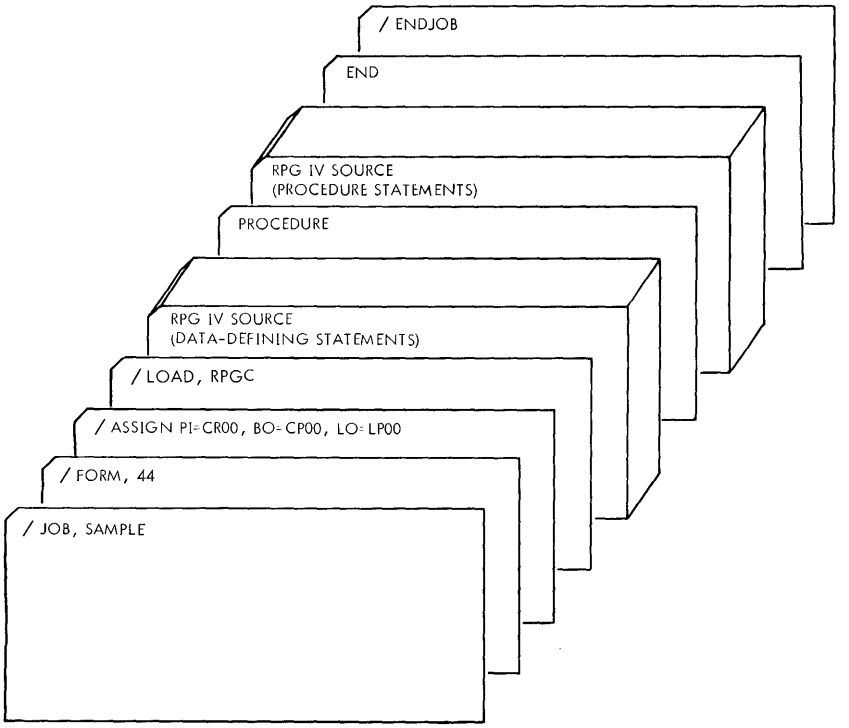
The PI, BO and LO devices are opened and rewound at the start of compilation, and are closed and updated at the end of compilation.



VTII-1539

Figure 4-2. MOS Job Stream for Compiling an RPG IV Program

A sample job stream for a VORTEX RPG compilation is shown in figure 4-3.



VTII-1540

Figure 4-3. VORTEX RPG IV Compilation

HARDWARE OPERATION

Stand-Alone Hardware Operation for RPG IV Compilation

To compile a program in RPG IV:

a. Place the compilation deck (figure 4-1) in the input hopper of the card reader with the binary card loader at the front of the deck.

b. Turn on and ready the card reader.

c. Turn on and ready the line printer. (continued)

d. Turn on and ready the card punch. Ensure that there are blank cards in the hopper. If the visual punch station is empty, insert a card into it as follows:

   (1) Place a card in the auxiliary feed slot.
   (2) Clear all registers.
   (3) Set the instruction register to 0100131.
   (4) Set RESET (not on V73 computer).
   (5) Press STEP (for V73, 620/f, and 620/f-100 computers, ensure computer is in the step mode and press START).
   (6) Reset REPEAT (not on V73 computer).

e. If it has not already been done, key in the 19-word card bootstrap loader (appendix D). Once done, this step can be omitted.

f. Clear the A, B, X, and instruction registers.

g. Set the P register to 000130.

h. Press RESET or SYSTEM RESET.

i. Press RUN (for V73, 620/f, and 620/f-100 computers, ensure computer is in the run mode and press START). The cards should start to move through the card reader.

If the compilation is successful, no further manual operation is required. When the END statement is reached, the computer stops in STEP mode with the A, B, and X registers cleared and the P register set to 000130.

Remove the object deck from the output hopper of the card punch and the program listing from the line printer.

Hardware Operation for RPG IV Compilation (MOS Version)

For procedures on operating the hardware for RPG IV compilation (MOS version), refer to the Varian 620 Master Operating System Reference Manual (98 A 9952 09x).

Hardware Operation for RPG IV Compilation (VORTEX Version)

For procedures on operating the system for RPG IV compilation (VORTEX version), refer to the VORTEX Reference Manual (98 A 9952 10x).

COMPILATION ERRORS

Serious compilation errors and irrecoverable I/O errors detected by the stand-alone version of RPG IV halt the computer. The following subsection describes such errors as

well as the recommended corrective action. The MOS version of RPG IV does not halt the computer but, instead, logs the errors and returns to the MOS executive (refer to the MOS reference manual for additional errors relating to I/O, loading, etc.). Similarly, the VORTEX version, upon detecting a serious error, logs and error message and returns control to the executive.

Language errors introduced through programming do not abort the compiler but, instead, produce diagnostic and error messages (see Language Errors).

Compilation Error Halts

Any of the following conditions stops compilation. To correct the error, use the recovery procedure indicated.

Card reader malfunction: This is indicated by an instruction register value of 000007. To recover:

a. Press the START button on the card reader.

b. Press RESET or SYSTEM RESET on the computer.

c. Press RUN on 620/L and 620/L-100 computers (for V73, 620/f, and 620/f-100 computers, ensure computer is in the run mode and press START).

Card punch malfunction: This is indicated by an instruction register value of 000031. To recover:

a. Add cards to the card hopper if it is empty.

b. If the visual punch station is empty, put a blank card in the auxiliary feed slot and set SENSE switch 1 on the computer.

c. Press RESET or SYSTEM RESET on the computer.

d. Press RUN on 620/L and 620/L-100 computers (for V73, 620/f, and 620/f-100 computers, ensure computer is in the run mode and press START).

END card found before the PROCEDURE card: This is indicated by an instruction register value of 0131 and 01 in each of the A, B, and X registers. To recover, remove the compilation deck, reassemble it correctly, and restart.

Program requires more memory than available: This is indicated by an instruction register value of 000131 and 0177777 (i.e., – 1) in each of the A, B, and X registers. There is no recovery. Run the program on a system having more memory, or restructure the program into smaller segments.

Excessive table size 0 This error is indicated by instruction register value of 00144. There is no recovery. The user should investigate alternatives for reducing his tables to an acceptable size for his computer memory.

Compilation Error Messages (MOS Version)

Error messages are listed followed by the cause.

Message	Cause
NO PROCEDURE CARD	An END card detected prior to a procedure card.
MEMORY FULL	a. Program requires more memory than is available. b. Table declaration is too large.

Compilation Error Messages (VORTEX Version)

The diagnostic messages produced during compilation are the same as those described in the following subsection. Fatal compilation errors or irrecoverable I/O errors during compilation cause and error message (see below) to be posted to the LO device, and the compilation terminated. These error messages are as follows:

RP01, NNN	I/O error
RP02, NNN	End of file error
RP03, NNN	End of device error
RP04	End card error (End card encountered before procedure card)
RP05	Available memory exceeded

Where NNN is the logical unit number on which the error occurred.

Language Errors

Any of the conditions tabulated in appendix C causes an error message to be printed on the program listing. In addition, there is an arrow pointing to the location of the error as a diagnostic aid. An error message and arrow are shown as follows:

```
                (1,4    'PAGE)
    SYNTAX
```

Compilation continues so that the program listing is complete. Thus, all listed errors can be detected and corrected on one pass.

To recover from these errors, correct the program statements containing the errors and recompile the program. Discard any object deck produced from a compilation containing errors.

LOADING AND EXECUTING AN RPG IV PROGRAM

STAND-ALONE VERSION DECK

The card deck preparation for loading and executing an RPG IV program is shown in figure 4-4 and comprises, in order:

a. Binary card loader (the same as for compilation)

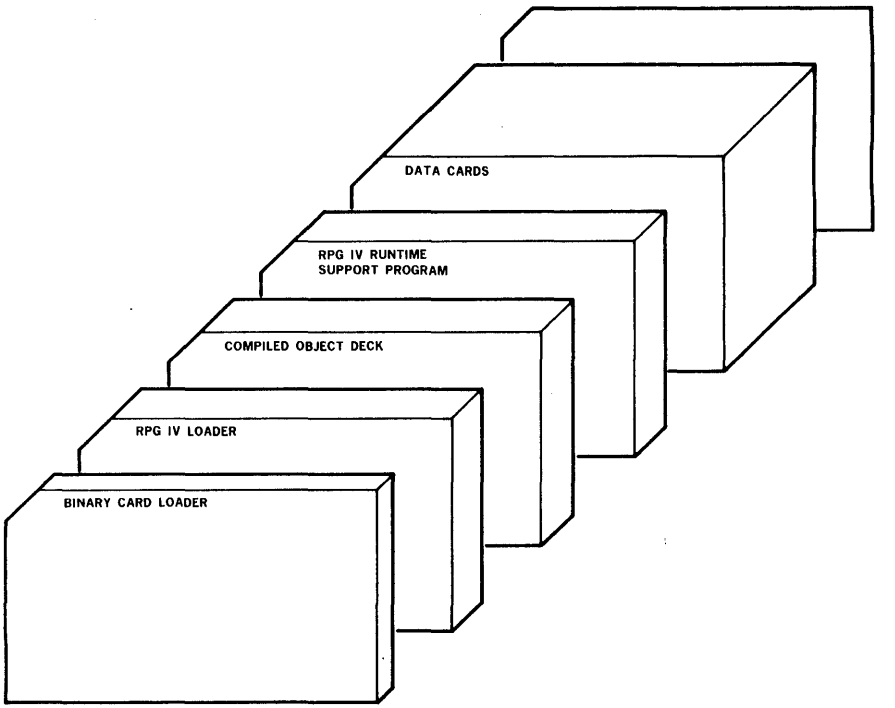
b. RPG IV loader

c. The compiled object module (deck) resulting from the compilation

d. RPG IV runtime support

e. Data cards as required by the program

f. Last card (/* in columns 1 and 2) as required by the program



VTII-1012

Figure 4-4. Runtime Deck for Proceeding RPG IV Data (Stand-Alone Version)

MOS VERSION DECK

The MOS RPG IV runtime/loader reads object records from the Binary Input file (BI) and logs any errors on the List Output file (LO). A sample job stream for an MOS load and execution is shown in figure 4-5 and comprises, in order:

a. An optional job card to identify jobs in the input stream.

b. An optional forms card if other than the default value for number of lines per page is desired.

c. An optional assignment card if assignments other than the default peripheral assignments are required, or if assurance of additional assignments is desired.

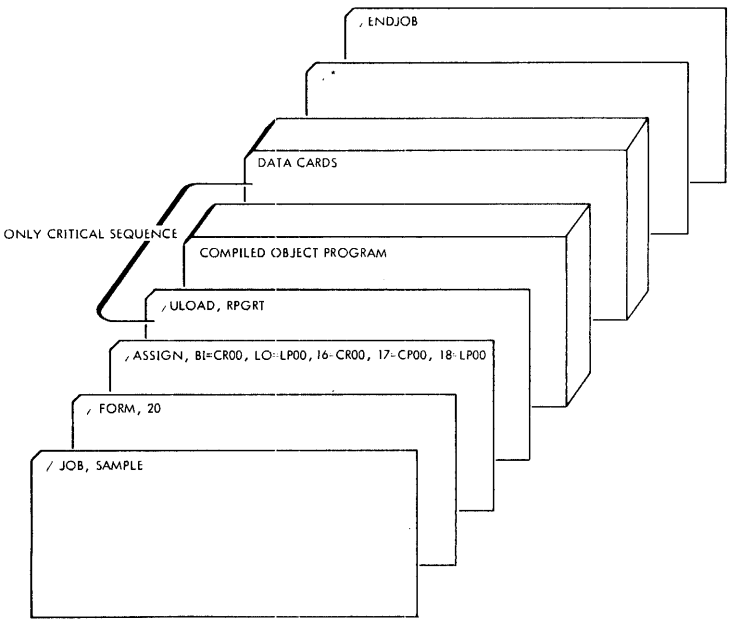
d. An unconditional load card to direct the loading of the RPG IV runtime/loader from the system file.

e. The RPG IV object program.

f. Optional data cards as required by the program.

g. An optional last card (/* in columns 1 and 2) indicated as required by the program.

h. An optional end-of-job card to separate jobs in the input stream.



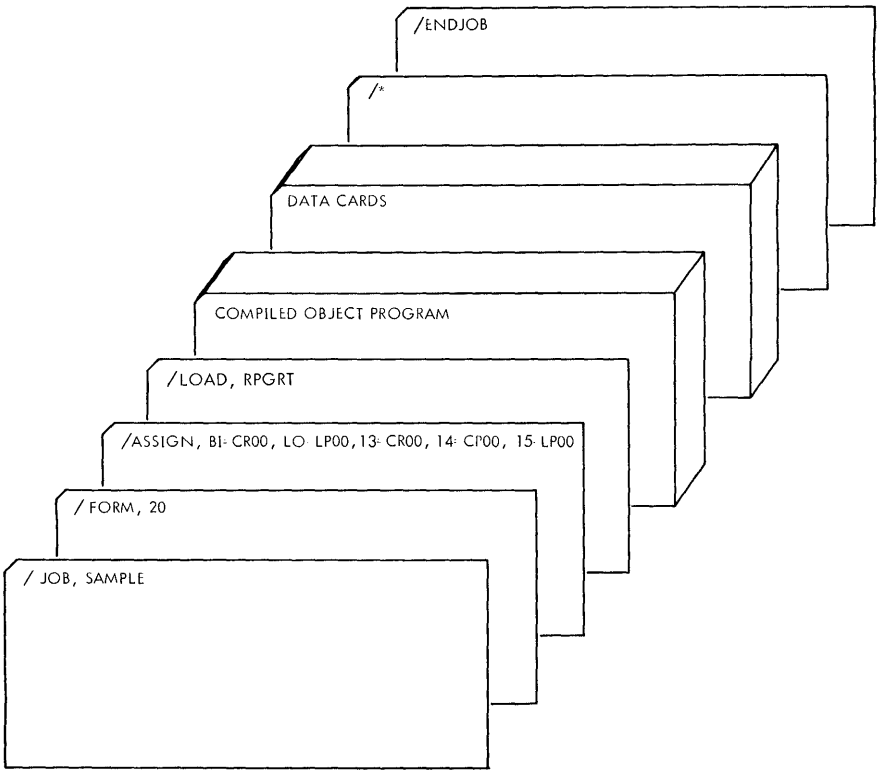
VTII-1541

Figure 4-5. MOS Job Stream for Loading and Executing an RPG IV Program

VORTEX VERSION DECK

The VORTEX RPG IV runtime/loader is scheduled from the background library at level 0 by the /LOAD,RPGRT directive. The runtime/loader will assume the RPG object program is on the Binary Input device (BI) and will read and execute it. If the load directive contains the name of a program to be loaded, as in /LOAD, RPGRT, NAME the runtime/loader will assume the program " NAME" is in the background library and will load and execute it. A RPG object program may be " catalogued" in the background library by creating a directory entry and allocating file space with FMAIN, and copying the object program into this file with IOUTIL.

A sample job stream for a VORTEX RPG load and execution is shown in figure 4-6.



VTII-1542

Figure 4-6. VORTEX Job Stream for Loading and Executing an RPG IV Program

LOADING ERRORS

Stand-Alone

To operate the hardware for the stand-alone RPG IV program, follow the directions given for compilation, replacing the compiler card deck with the runtime deck.

Any of the following error messages can appear on the line printer output upon detection of the corresponding loading error. Except for the missing subroutines error, any of these errors causes a halt in the loading after the message is printed.

PROG TOO BIG: This message indicates that the object deck requires more memory than is available in the system. There is no recovery. Run the program on a system having more memory, or restructure the program into smaller segments.

INVALID OBJECT DECK: This message indicates that the loader encountered a directive or format not conforming to those of a normal compiler output. There is no recovery. Recompile the program. If the error persists, check the card punch and recompile.

CHECKSUM ERROR: This message indicates that the last card read produced a checksum error. Back up the erroneous card and press READ to reread it. If the error persists, check the card punch and recompile.

CARD SEQUENCE ERROR: This message indicates that the cards in the object deck are out of sequence. Correct the sequence and reload. (The eight-bit sequence field on the object deck cards is in rows 6 through 9 of column 1 and rows 12 through 1 of column 2, where row 1 of column 2 is the least significant bit.)

PROG NOT EXECUTABLE: This message indicates that the program contains an error that would prevent output of the correct results. There is no recovery. Correct the program and recompile.

MISSING SUBROUTINE: This message, followed by the names of the programs that are missing, indicates that there is a CALL statement reference to a subprogram not included in the runtime deck. Loading continues and the object program is executed up to the call to the missing program. At this point execution halts. There is no recovery. Insert the missing program in the program (appendix E) and recompile.

MOS Version

The only loading error message directly output by the RPG IV loader is MISSING SUBROUTINE. Its definition is identical to the stand-alone version. Additional loading error messages are logged by the MOS Executive as a result of faults detected by the RPG IV loader (see MOS Reference Manual for status and error messages of the system loader).

oading Errors (VORTEX Version)

he VORTEX RPG IV runtime/loader, upon detecting an error while loading an object rogram, will post an error message on logical unit 15 and terminate further activity, eturning control to the executive. These error messages are as follows:

RT01, NNN	I/O error
RT02, NNN	End of file error
RT03, NNN	End of device error
RT04	Program too big
RT05	Invalid object record
RT06	Checksum error
RT07	Sequence error
RT08	Program not executable

vhere NNN is the logical unit number on which the error occurred.

An additional error message which may occur at load time is:

RT10, xxxxxx

where xxxxxx is a missing subroutine name. After this message is posted, loading will continue. If an attempt is made to execute one of the missing subroutines control will be returned to the executive.

EXECUTION (RUNTIME) ERRORS

Stand-Alone

Any of the following conditions stops execution of the object program. To correct the error, use the recovery procedure indicated, if one is possible.

STOP statement encountered: This is indicated by an instruction register value of 000001 and an operator message on the line printer. To continue execution, press START or RUN on the computer.

Missing CALL subroutine: This is indicated by an instruction register value of 000002. There is no recovery. Insert the missing subroutine in the program deck and recompile.

Worklist overflow: This is indicated by an instruction register value of 000003. This results from an overflow on an internal worklist stack during complicated arithmetical manipulations, etc. There is no recovery from this rare condition. Run the program on a system having more memory, or recode the lengthy expression into smaller subexpressions.

Card reader malfunction: This is indicated by an instruction register value of 000007.

To recover:

a. Press the START button on the card reader.

b. Press RESET or SYSTEM RESET on the computer.

c. Press RUN on 620/L and 620/L-100 computers (for V73, 620/f, and 620/f-100 computers, ensure computer is in the run mode and press START).

Card punch malfunction: This is indicated by an instruction register value of 000031.

To recover:

a. Add cards to the card hopper if it is empty.

b. If the visual punch station is empty, put a blank card in the auxiliary feed slot and set SENSE switch 1 on the computer.

c. Press RESET or SYSTEM RESET on the computer.

d. Press RUN on 620/L and 620/L-100 computers (for V73, 620/f, and 620/f-100 computers, ensure computer is in the run mode and press START).

MOS Version

The RPG IV runtime program under MOS detects and outputs the two error messages described below. Additional faults dealing with I/O are detected by IOCS and logged accordingly (see MOS Reference Manual for status and error messages of I/O control). All runtime error messages are followed by the program being aborted and control returned to MOS.

INVALID RPG CALL TO: This message is printed when a CALL statement contains invalid arguments (i.e., too few, too many, or the wrong type).

RPG WORKLIST OVERFLOW: This message is printed when the internal worklist stack space is exceeded during arithmetic manipulation, etc. To correct this condition, run the program on a system having more memory, or recode the lengthy expression into smaller subexpressions.

VORTEX Version

The VORTEX RPG IV runtime/loader, upon detecting an error while executing an object program, will post an error message on logical unit 15 and terminate further activity, returning control to the executive. The error messages are as follows:

RT01, NNN	I/O error
RT02, NNN	End of file error
RT03, NNN	End of device error
RT08	Program not executable
RT09	Work list overflow
RT10, xxxxxx	Invalid call to subroutine

where NNN is the logical unit number on which the error occurred.

xxxxxx is the subroutine name.

RT10 errors may be caused by invalid parameters in the CALL, such as invalid RPG unit, invalid VORTEX logical unit, access method or mode ‡ 1, or by an attempt to open an already opened file.

SECTION 5 - SAMPLE RPG IV PROGRAM

CALENDAR PROGRAM

This program will print a calendar for any year between 1582 (when the Gregorian calendar was adopted) and 4901. The year 4901 is a terminal date for the Gregorian calendar because by that year the astronomical year will be out of step by one day (this occurs approximately every 3,000 years).

Note

A year is a leap year if:

a. It is evenly divisible by 4, but *not* by 100
1900 is not; 1904 is.

or

b. It is evenly divisible by 400
1900 is not; 2000 is.

The day of the week can be calculated by use of the formula:

DW = ([2.6*M] + DM + B + [B/4] + [C/4]-2C)MOD7

where

DW = Day of the week (0 = Sun., ..., 6 = Sat.)

M = Month (Jan. = 11, Feb. = 12 of previous year and Mar. = 1, ..., Dec. = 10 of current year

DM = Day of the month (1, ..., 31)

B = Last two digits of the year

C = First two digits of the year

( )MOD7 = Remainder after dividing by 7 until ( ) is less than 7

The months are printed in four rows of three columns. The program initialization (lines 10 through 30) inputs the heading cards. Loop 1 (lines 40 through STOP) reads in a year, prints the calendar for that year, and, on reading the last card (/*), prints " end of JOB' and then stops.

Loop 2 (lines 80 through 190) is inside loop 1, and prints four rows of three months each (rows numbered 0 through 3). Inside loop 2 is loop 3 (lines 110 through 190) which prints six lines (one for each week, see Jan. 1971). Loop 4 (lines 130 through 190), inside loop 3, creates the line of the week for each column (numbered 1, 2, 3). The last loop is 5 (lines 150 through 190) which calculates the day of the week for each day of the month and inserts it into the line. If the day is not a Saturday, the line is shifted left one position, and the next day is tried.

This method has a very slow execution rate, but it demonstrates the use of loops within loops, and of overlapping fields (SO, SP). The figures that follow illustrate this program. Figure 5-1 is the program, 5-2 is the flowchart, 5-3 is the input data, and figure 5-4 is the printed output.

```
VARIAN RPG IV SOURCE LISTING

        TABLE   MO        (5)
                MONG      (1,72)
        RECORD  IN
                YIN       (1,4)
                YEAR      (1,4,0)
                C         (1,2,0)
                B         (3,4,0)
                M         (5,6,0)
                DW        (7,9,0)
                SO        (10,12,0)
                SP        (10,13,1)
                SQ        (14,15,0)
        RECORD  LINE
                YOUT      (35,38),B
                COL1      ( 1,71),B
                COL2      (26,71),B
                COL3      (51,71),B
                COL4      (54,71),B
                D         (70,71,0),B,Z
        TABLE T           (12)
                LM        (1,2,0)
                DM        (3,4,0)
        PROCEDURE
        10                MOVE 1,I
        20                READ CARD LINE
                          MOVE COL1,MONG(I)
                          COMPUTE I=I+1
                (I<6)     GO TO 20
                          MOVE ' ',COL1
        30                MOVE 31,LM(1),LM(3),LM(5),LM(7),LM(8),LM(10),LM(12)
                          MOVE 30,LM(4),LM(6),LM(9),LM(11)
        40                READ CARD IN
                (#LC)     PRINT $C1
                (#")      STOP 'END OF JOB'
        50                MOVE YIN, YOUT
                          PRINT $C1,LINE
        60                COMPUTE SP=B/4
                          "       SQ=C/4
                          MOVE 28,LM(2)
                ((B=4*SO) AND ((B><0) OR (C=4*SQ)))MOVE 29,LM(2)
        70                COMPUTE ROW=-1
        80                COMPUTE ROW=ROW+1
                (ROW>3) GO TO 40                                    (continued)
```

VTII-1179

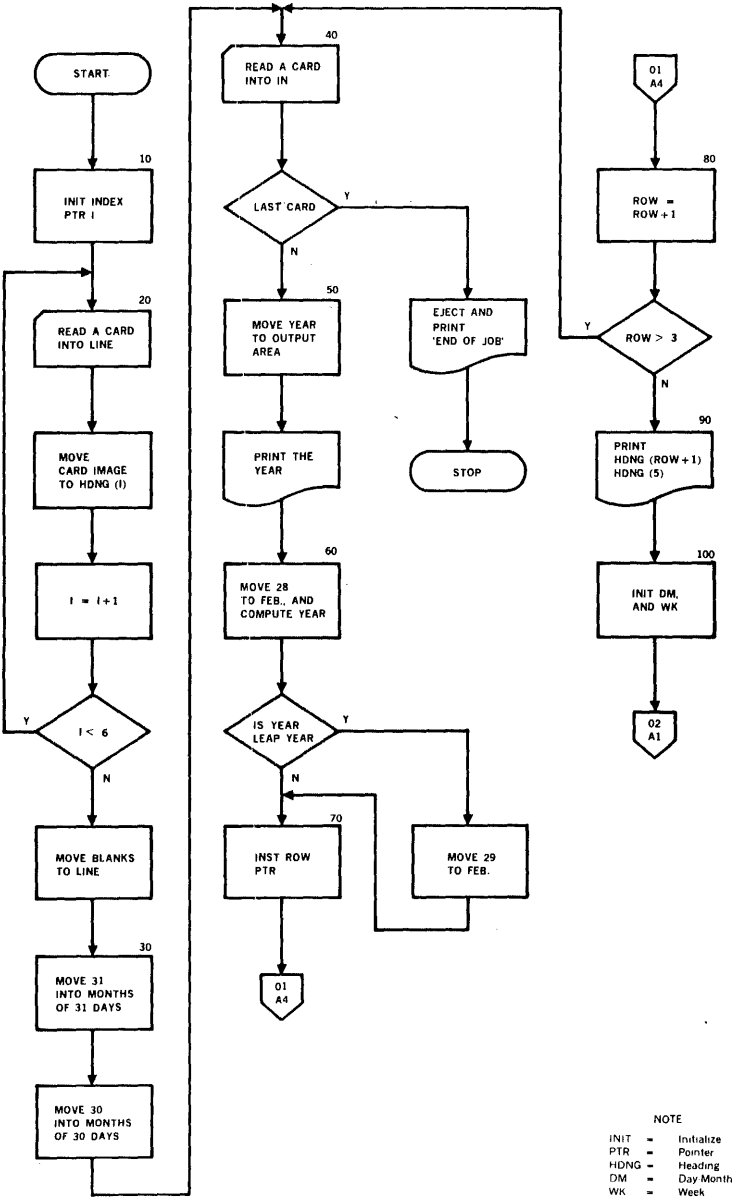
Figure 5-1. Calendar Program

```
VARIAN RPG IV SOURCE LISTING

      90              COMPUTE ROW1=ROW+1
                      MOVE HDNG(ROW1),COL1
                      PRINT $A3,LINE
                      MOVE HDNG(5),COL1
                      PRINT LINE
     100              MOVE 0,DM(1),DM(2),DM(3),WK
     110              COMPUTE WK=WK+1
              (WK>6)  GO TO R0
     120              MOVE 1,COL
     130              COMPUTE M=(3*ROW)+COL
     140              MOVE COL2,COL1
     135      (M=1)   COMPUTE YEAR=YEAR-1
              (M=3)   "       YEAR=YEAR+1
     150              MOVE COL4,COL3
     160              COMPUTE DM(COL)=DM(COL)+1
     170              COMPUTE N=M-2
              (N<1)   "       N=N+12
                      "       SP=2.6*N-0.2+DM(COL)+B-2*C+105
                      "       SP=SO+B/4
                      "       DW=SO+C/4
                      "       SP=DW/7
                      "       DW=DW-7*SO
                      COMPUTE SQ=LM(M)-DM(COL)+1
     180      (SQ>0)  MOVE DM(COL),D
     190      (DW<6)  GO TO 150
                      COMPUTE COL=COL+1
              (COL<=3)GO TO 130
                      PRINT LINE
                      GO TO 110
              END
```

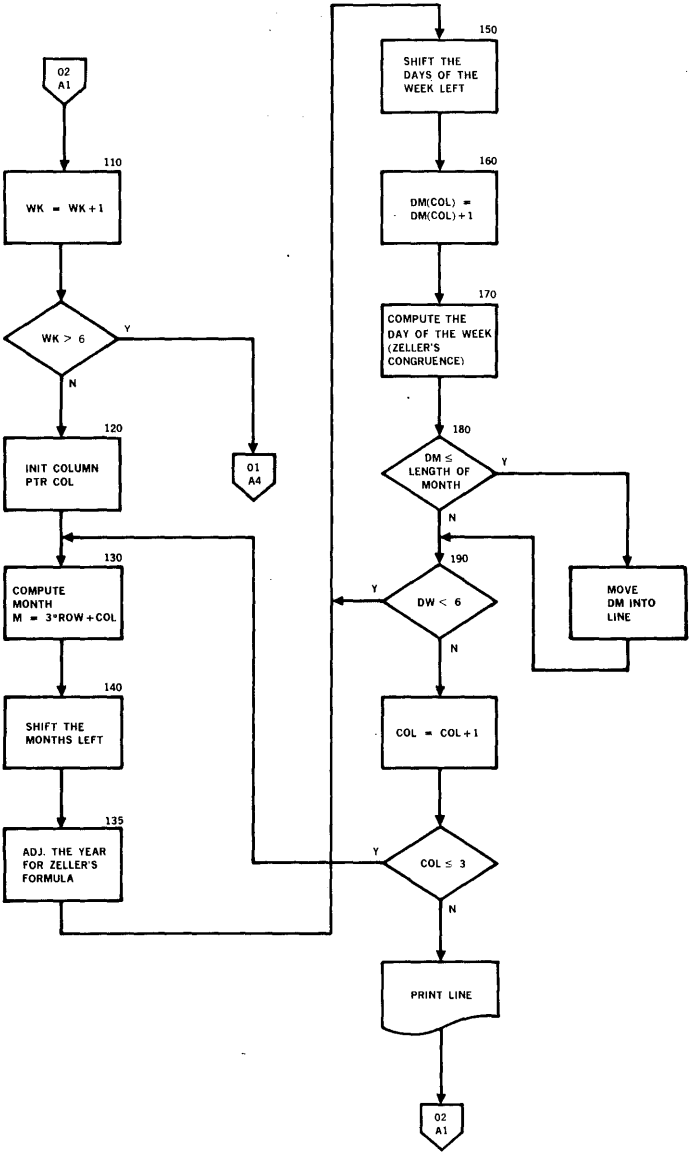
VTII-1180

Figure 5-1. Calendar Program *(continued)*



VTI2-0367

Figure 5-2. Flowchart for Calendar Program



VTI2-0368

Figure 5-2. Flowchart for Calendar Program *(continued)*

```
      JANUARY                  FEBRUARY                  MARCH                1
      APRIL                    MAY                       JUNE                 2
      JULY                     AUGUST                    SEPTEMBER            3
      OCTOBER                  NOVEMBER                  DECEMBER             4
  S  M  T  W  T  F  S      S  M  T  W  T  F  S      S  M  T  W  T  F  S  5
1971
VTII-1181
```

Figure 5-3. Input Data

```
                                    1971

      JANUARY                  FEBRUARY                  MARCH
  S  M  T  W  T  F  S      S  M  T  W  T  F  S      S  M  T  W  T  F  S
                  1  2         1  2  3  4  5  6         1  2  3  4  5  6
  3  4  5  6  7  8  9      7  8  9 10 11 12 13      7  8  9 10 11 12 13
 10 11 12 13 14 15 16     14 15 16 17 18 19 20     14 15 16 17 18 19 20
 17 18 19 20 21 22 23     21 22 23 24 25 26 27     21 22 23 24 25 26 27
 24 25 26 27 28 29 30     28                       28 29 30 31
 31

      APRIL                    MAY                       JUNE
  S  M  T  W  T  F  S      S  M  T  W  T  F  S      S  M  T  W  T  F  S
                  1  2  3                  1                  1  2  3  4  5
  4  5  6  7  8  9 10      2  3  4  5  6  7  8      6  7  8  9 10 11 12
 11 12 13 14 15 16 17      9 10 11 12 13 14 15     13 14 15 16 17 18 19
 18 19 20 21 22 23 24     16 17 18 19 20 21 22     20 21 22 23 24 25 26
 25 26 27 28 29 30        23 24 25 26 27 28 29     27 28 29 30
                          30 31

      JULY                     AUGUST                    SEPTEMBER
  S  M  T  W  T  F  S      S  M  T  W  T  F  S      S  M  T  W  T  F  S
                  1  2  3      1  2  3  4  5  6  7                  1  2  3  4
  4  5  6  7  8  9 10      8  9 10 11 12 13 14      5  6  7  8  9 10 11
 11 12 13 14 15 16 17     15 16 17 18 19 20 21     12 13 14 15 16 17 18
 18 19 20 21 22 23 24     22 23 24 25 26 27 28     19 20 21 22 23 24 25
 25 26 27 28 29 30 31     29 30 31                 26 27 28 29 30

      OCTOBER                  NOVEMBER                  DECEMBER
  S  M  T  W  T  F  S      S  M  T  W  T  F  S      S  M  T  W  T  F  S
                  1  2         1  2  3  4  5  6                  1  2  3  4
  3  4  5  6  7  8  9      7  8  9 10 11 12 13      5  6  7  8  9 10 11
 10 11 12 13 14 15 16     14 15 16 17 18 19 20     12 13 14 15 16 17 18
 17 18 19 20 21 22 23     21 22 23 24 25 26 27     19 20 21 22 23 24 25
 24 25 26 27 28 29 30     28 29 30                 26 27 28 29 30 31
 31

END OF JOB
VTII-1182
```

Figure 5-4. Printed Output

APPENDICES

INDICATOR CHART

Where Used	Turned on by:	Turned off by:
GENERAL INDICATORS #1 THROUGH #99		
General purpose	SET statement	SET statement
Sequence-checking	Fields out of sequence	Fields in sequence
Auditing	Successful audit	Unsuccessful audit
Record selection	Meeting criteria	Not meeting criteria
Table overflow	Out-of-range table reference	SET statement
CONTROL BREAK INDICATORS #C1 THROUGH #C10		
Control breaks	Direct updating that changes this or any higher-level control field (except when #F is on)	Direct updating that does not change this control field *or* direct updating that changes this or any higher-level control field when #F is on
SPECIAL INDICATORS		
Repeat verb or condition of previous statement **#"**	Execution of previous statement	Nonexecution of previous statement
Table-searching **#G, #L, #E**	LOOKUP statement	LOOKUP statement
Result of computation **#P, #Z, #M**	COMPUTE statement	COMPUTE statement
First control break **#F**	First direct updating of any control field	Subsequent direct updating of any control field

(continued)

INDICATOR CHART (continued)

Where Used	Turned on by:	Turned off by:
Computational Overflow **#X1**	Numeric value placed in a field that cannot hold it *or* any computation (including subscript and relational expressions) whose value exceeds 99,999,999,999	SET statement
Mode error **#X2**	Attempted use of a field of wrong mode (numeric/alphanumeric)	SET statement
Last card **#LC**	READ CARD statement after /* card	SET statement
Page overflow **#OV**	PRINT statement in which the line count = 44 *or* one that causes a skip to channel 7 or /FORM	PRINT statement that causes a skip to channel 1

NOTE

The initial state of #OV is *on*; that of all other indicators is *off*.

COLLATING SEQUENCE AND CHARACTER REPRESENTATION

Collating Sequence	Graphic	Standard 029 Keypunch	026 Keypunch
1	(Blank)	(no punch)	(no punch)
2	[	12-2-8	12-5-8
3	.	12-3-8	12-3-8
4	<	12-4-8	12-6-8
5	(	12-5-8	0-4-8
6	+	12-6-8	12
7	!	12-7-8	7-8
8	&	12	12-7-8
9	!	11-2-8	11-2-8
10	$	11-3-8	11-3-8
11	*	11-4-8	11-4-8
12	)	11-5-8	12-4-8
13	;	11-6-8	11-6-8
14	¬	11-7-8	0-6-8
15	-	11	11
16	/	0-1	0-1
17	,	0-3-8	0-3-8
18	%	0-4-8	11-2-8
19	¬	0-5-8	2-8
20	>	0-6-8	6-8
21	?	0-7-8	12-2-8
22	:	2-8	5-8
23	#	3-8	0-7-8
24	@	4-8	0-2-8
25	'	5-8	4-8
26	=	6-8	3-8
27	"	7-8	0-5-8
28	A	12-1	12-1
29	B	12-2	12-2
30	C	12-3	12-3
31	D	12-4	12-4
32	E	12-5	12-5
33	F	12-6	12-6
34	G	12-7	12-7
35	H	12-8	12-8
36	I	12-9	12-9
37	J	11-1	11-1
38	K	11-2	11-2
39	L	11-3	11-3
40	M	11-4	11-4
41	N	11-5	11-5

(continued)

COLLATING SEQUENCE AND CHARACTER REPRESENTATION (continued)

Collating Sequence	Graphic	Standard 029 Keypunch	026 Keypunch
42	O	11-6	11-6
43	P	11-7	11-7
44	Q	11-8	11-8
45	R	11-9	11-9
46	]	0-2-8	11-5-8
47	S	0-2	0-2
48	T	0-3	0-3
49	U	0-4	0-4
50	V	0-5	0-5
51	W	0-6	0-6
52	X	0-7	0-7
53	Y	0-8	0-8
54	Z	0-9	0-9
55	0	0	0
56	1	1	1
57	2	2	2
58	3	3	3
59	4	4	4
60	5	5	5
61	6	6	6
62	7	7	7
63	8	8	8
64	9	9	9

COMPILATION ERROR MESSAGES

Message	Location of Arrow	Error
INDICATOR	Invalid character	Character other than # at beginning of assumed indicator symbol
	Invalid character	Invalid indicator character
INVALID	End of section table field statement containing KEY	More than one key field for a table
	Last character of table name	DELETE references a LIFO table
LABEL	Last digit of statement number	Duplication of statement numbers
	None -- message after END	Undefined statement number
LITERAL	Closing quotation mark	No character between opening and closing single quotation marks
	Last nonblank character	End of line found before closing single quotation mark
NAME	Last character of name	Duplication of table or record names
	Seventh character of name	More than six characters in a name
	Period	Field name followed by period or qualified name where nonfield name is required
	Last character of field name	Field name of a qualified name not defined in a record or table

COMPILATION ERROR MESSAGES (continued)

Message	Location of Arrow	Error
NAME *(continued)*	Left parenthesis of subscript	Subscripted record or record field name
	Last character of name	Invalid reference to name (e.g., a field name where a record name is required)
	Last character of table name	LOOKUP references LIFP table
	Right parenthesis of subscript	LOOKUP contains a subscript
	Right parenthesis of subscript	Subscripted name in CALL argument list
RELATIONAL	Last character of relational expression	Relational expression contains two literals, or attempts to compare an alphanumeric field to a constant or expression
SIZE	Last digit of boundary	Zero field boundary
	Last digit of field-end spec	Field width not positive
	Last digit of number	Numeric portion of indicator quantity zero or too large
	Digit causing overflow	Accumulation of integer quantity causes arithmetic overflow
	Last digit of selection column	Record selection column zero or > 255
	Last digit of statement number	Statement number zero or > 9999
	Fifteenth digit	> 14 digits before decimal point
	Tenth digit	> 9 digits after decimal point

COMPILATION ERROR MESSAGES (continued)

Message	Location of Arrow	Error
SIZE *(continued)*	Last digit of number	Numeric portion of PRINT zero or > 7
SYNTAX	Invalid character	First character of name not alphabetic
	Character following name	Missing (in field definition
	Character following last digit of field-end specification or fractional length	Missing) in field definition
	Character following subscript	Missing) on subscript
	Invalid character	First character of assumed integer not numeric
	First character of statement	Field statement found before record or table statement
	Invalid character	Invalid selection identification
	Invalid character	Nonblank character in statement after meaningful specifications complete
	Invalid character	Editing character in definition of a nonnumeric field
	Invalid character	First nonblank character after TABLE in a table statement not (
	Nonblank character following number of entries	Missing) after number of entries in table statement
	Invalid character	Nonblank character after PROCEDURE
	Nonblank character following conditional expression	Missing) in conditional expression

COMPILATION ERROR MESSAGES (continued)

Message	Location of Arrow	Error
SYNTAX *(continued)*	Nonblank character following field name	Missing = in COMPLETE statement
	Column 69 of SET statement	Missing indicator list in SET statement
	Nonblank character following position meant for missing item	No condition, ON, or OFF in SET statement
	Nonblank character following last statement number in list	Missing) in indexed GO TO statement
	Column 69 of statement	Missing to-field in statement that requires one
	Column 69 of LOOKUP statement	Missing index field in LOOKUP statement
	Nonblank character following expression	Missing) on expression that began with (
	Nonblank character following first operand	Missing relational operator after first operand of relational expression
	Invalid character	First character of assumed decimal constant neither digit nor period
	Column 69 of PRINT statement	Missing argument list in PRINT statement
	First character not matching the sequence C A R D	Keyword CARD missing in READ CARD statement
	Nonblank character following CARD	Missing delimiter after READ CARD statement character string

CARD BOOTSTRAP LOADER

Using the data entry switches on the computer front panel, enter the card bootstrap loader as follows:

Memory Address	Octal Contents		DAS Code	
000114	102530	BOOR	CIA	030
000115	004250		LRLA	8
000116	101130		SEN	0130,BOOS
000117	000122			
000120	001000		JMP	* - 2
000121	000116			
000122	102130	BOOS	INA	030
000123	055000		STA	0,1
000124	005144		IXR	
000125	001000		JMP	BOOU
000126	000131			
000127	000000	BOOT	DATA	PLD
000130	100230		EXC	0230
000131	101130	BOOU	SEN*	0130,BOOR
000132	000114			
000133	101630		SEN	0630,BOOT
000134	100127			
000135	001000		JMP	* - 4
000136	000131			

Clear the registers.

Load the B register with the upper boundary of a 4K memory module, thus delimiting the virtual memory for a given run; e.g., set the B register to 010000, 020000, etc., to indicate a virtual memory of 4K, 8K, etc., words. If the B register contains zero, the loader locates itself at the top of the memory starting with address 0x7660 (x = 0 for a 4K memory, 1 for an 8K memory, etc.).

Set the P register to 000130.

Put the RPG IV loader, followed by the compiler or runtime deck in the card reader.

Press SYSTEM RESET or RESET.

Press RUN or START. The bootstrap loader loads the binary card loader and transfers control to it for further loading.

CALL STATEMENT SUBROUTINE USAGE

STAND-ALONE VERSION

The stand-alone version of RPG IV provides for linking up to two user-written, DAS-coded subroutines. Once loaded, these subroutines can then be executed through the RPG IV CALL statement.

Linkage

The RPG IV loader and runtime programs both contain an empty table, labeled STAB, which is used to establish linkage to called subroutines. The table is initialized to blanks and zeros and can be overlayed at load time by the user to establish subroutine names and entry points. Figures E-1 and E-2 depict the actual code as it appears in the loader and runtime programs. The table format is:

Word 1	Number of subroutine names in the table (0, 1, or 2)
Words 2-4	Subroutine name in ASCII (preset to blanks)
Word 5	Pointer to subroutine entry address (preset to zero)
Words 6-8	Subroutine name in ASCII (preset to blanks)
Word 9	Pointer to subroutine entry address (preset to zero)

When the loader encounters an external reference in the object deck, it searches STAB for the name. If the name is in the table, a runtime CALL instruction is stored with the operand equal to the entry number of the subroutine name in STAB. The runtime interpreter uses this value to locate the beginning of any specified subroutine. If the name is not in the table, the error message MISSING SUBROUTINE is printed on the line printer (chapter IV).

```
                    *                   SUBROUTINE CALL TABLE (STAB)
                    *          THE FOLLOWING TABLE CONTAINS THE REQUIRED INFORMATION
                    *               TO ENABLE THE RPG IV LOADER TO LINK TO AN EXTERNAL
                    *                   SUBROUTINE WHICH IS REFERENCED BY A 'CALL'
                    *                        STATEMENT.  THE TABLE PROVIDES ROOM FOR
                                                 TWO ENTRIES.  EACH ENTRY CONTAINS TH
                                                      FOLLOWING DATA:
                    *
                    *                                      1.  NAME OF SUBROUTINE
                    *
                    *                                      2.  SUBROUTINE ENTRY ADDRESS
                    ORG  0770
000770  000002      STAB DATA 2          NUMBER OF ENTRIES
000771  120240           DATA '  '       SUBROUTINE NAME- 1ST ENTRY
000772  120240
000773  120240
000774  000000           DATA 0               SUBROUTINE ENTRY ADDRESS
000775  120240           DATA '  '       SUBROUTINE NAME- 2ND ENTRY
000776  120240
000777  120240           DATA 0               SUBROUTINE ENTRY ADDRESS
001000  000000
                    *
                    *               (DATA IN THE SUBROUTINE CALL TABLE MUST
                    *                    CORRESPOND IDENTICALLY TO THE DATA
                    *                         CONTAINED IN THE SUBROUTINE CALL
                    *                              TABLE (STAB) OF THE RPG IV
                    *                                   RUNTIME SUPPORT PROGRAM)
```

Figure E-1. STAB Table In Loader

```
                    *                   SUBROUTINE CALL TABLE (STAB)
                    *     THE FOLLOWING TABLE CONTAINS THE REQUIRED INFORMATION
                    *          TO ENABLE THE RPG IV RUNTIME SUPPORT PROGRAM TO LINK
                    *               TO AN EXTERNAL SUBROUTINE WHICH IS REFERENCED BY
                    *                    A 'CALL' STATEMENT.  THE TABLE PROVIDES
                    *                         ROOM FOR TWO ENTRIES.  EACH ENTRY HAS
                    *                              THE FOLLOWING DATA:
                    *
                    *                                   1. NAME OF SUBROUTINE
                    *
                    *                                   2. SUBROUTINE ENTRY ADDRESS
                    ORG  0721
000721  000002      STAB DATA 2              NUMBER OF ENTRIES
000722  120240           DATA '  '           SUBROUTINE NAME- 1ST ENTRY
000723  120240
000724  120240
000725  000000           DATA 0                   SUBROUTINE ENTRY ADDRESS
000726  120240           DATA '  '           SUBROUTINE NAME- 2ND ENTRY
000727  120240
000730  120240
000731  000000           DATA 0                   SUBROUTINE ENTRY ADDRESS
                    *
                    *                   DATA CONTAINED IN THIS TABLE MUST
                    *                        CORRESPOND IDENTICALLY TO THE DATA
                    *                             CONTAINED IN THE SUBROUTINE CALL
                    *                                  TABLE (STAB) OF THE RPG IV
                    *                                       LOADER PROGRAM
```

Figure E-2. STAB Table in Runtime

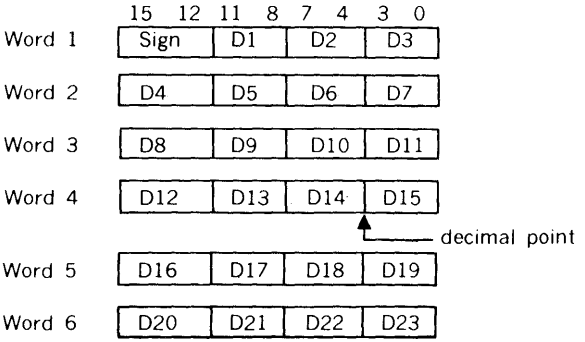
Arguments

To retrieve arguments from the CALL statement calling sequence, the called subroutine requires the service of runtime support routine GETR. GETR is called for each argument in the calling sequence. After each return from GETR, the A register contains a pointer to the first (next) argument in the calling sequence. The interpreter location counter is stepped past the argument on each call to GETR so that, after all of the arguments have been retrieved, the location counter is positioned for the interpreter to continue with the next interpretive instruction. The contents of location REND in the runtime program are negative after the last argument is fetched with GETR. Linkage to GETR can be made by a JMPM* 000177 instruction, and REND is at location 000017.

Arguments passed to called subroutines are one of two types: a numeric constant or a record, table, or field name. Numeric constants occupy six consecutive words of memory and have the form shown in figure E-3.

Record, table, and field names as arguments are accessed by a call to GETR also. Upon return, the A register indicates one less than the memory location that contains the byte address of the argument. To form a word address, the user is required to right-shift this value one position (e.g., LSRA 1).

After all arguments are processed and the subroutine has completed its function, control can be returned to the runtime interpreter by a JMP* 000220 instruction.



NOTES

a. A call to GETR returns the A register contents to word 1.

b. For a plus sign, bits 12 through 15 of word 1 equal a value of six. For a negative sign, they equal a value of seven.

c. The data fields (D1 through D23) are in 4-bit binary-coded decimal (BCD) each field can have a value of zero through nine.

Figure E-3. Word Formats of Numeric Constants

Coding

Subroutines to be used in CALL statements must be first coded in DAS assembly language and assembled with either the DAS 4A or DAS 8A assembler. The object output must be on punched cards. The assembly language source deck contains the following as the last seven statements of the program:

Card	Operation	Operand
1	ORG	Octal address of the first word for this entry in the subroutine call table (STAB) in the RPG IV runtime support program (value = 000721).
2	DATA	Subroutine name enclosed between apostrophes. The subroutine name (exclusive of apostrophes) must correspond exactly to the name used in the CALL statement of the RPG IV program.
3	DATA	Octal address of the subroutine entry location.
4	ORG	Octal address of the first word of the corresponding entry in the subroutine call table (STAB) of the RPG IV loader program (value = 000770).
5	DATA	Subroutine name enclosed between apostrophes. The subroutine name must correspond exactly to the name used in the CALL statement of the RPG IV program.
6	DATA	Octal address of the subroutine entry location.
7	END	

Card 7 is the END statement of the source deck itself; there is no operand after assembly. The three final object cards contain the following data:

a. Object data from cards 1, 2, and 3 above

b. Object data from cards 4, 5, and 6 above

c. Normal card object end card

Remove the next-to-last card and place it in the front of the last card of the RPG IV loader object deck. Place the remaining object cards of the assembled routine at the end of the RPG IV runtime support deck with the last card of that deck removed.

Subroutines such as these, which are called at runtime by an RPG IV program, reside in that part of upper memory not otherwise used by RPG IV. This area can be determined empirically by the following procedure:

a. Initialize each word of memory to some specific value using the 620 AID II program and the command:

   1. Ia,b,c, (initialize locations a through b to c)

b. Load and run the RPG IV program with the subroutine(s) omitted.

c. When RPG IV attempts to execute a missing subroutine, the computer halts with 000002 in the instruction register. Using AID II and the command:

   2. Sa,b,c, (search locations a through b to c)

scan the computer memory for the value to which memory was previously initialized. That block of memory below the resident card binary loader (which is filled with the value to which memory was previously initialized) is then available for containing the subroutine(s) whose name(s) appear in CALL statements in the RPG IV program.

Figure E-4 shows sample coding for a DAS subroutine and illustrates many of the details described in the preceding pages.

MOS VERSION

The MOS version of RPG IV allows for inclusion of DAS-coded subroutines to augment an RPG IV program. Such subroutines must be assembled into the runtime/loader program; hence, linkages are handled through the assembly process. Subroutines can be included after the end of the runtime/loader code; they require an entry into the STAB table (figures E-1 and E-2).

Since the loader and the runtime programs are combined under MOS, there is only one STAB table. Its size is variable and the user can add as many special-purpose subroutines to his system as needed. Refer to RPG IV runtime/loader source listing MOS version (Varian Software Parts Catalog, document number 98 A 9949 060).

Argument-processing is the same as in the stand-alone version except that GETR must be called directly (i.e., JMPM GETR) and control must be returned directly to the interpreter (i.e., JMP INT).

VORTEX VERSION

The method of inclusion and use of DAS-coded subroutines in the RPG runtime/loader program is the same as that described for the MOS version.

```
                    *
                    *     PARTIAL SAMPLE OF AN RPG IV SUBROUTINE
                    *     DAS CODED FOR STAND-ALONE VERSION
                    *     SUBROUTINE SHIFTS AN RPG NUMERIC FIELD RIGHT OR LEFT 'N' PLAC
                    *     ENTER:  CALL SHIFT,F,D,N
                    *             WHERE F = FIELD NAME
                    *                   D = DIRECTION (+ FOR RIGHT, - FOR LEFT
                    *                   N = NUMBER OF PLACES IN DECIMAL (9 MAXIMUM)
                    *     EXIT:   FIELD SHIFTED
                    *
          000177    GETR EQU       0177
          000017    REND EQU       0017
          000220    INT  EQU       0220
                    *
016000              SHIFT ORG      016000
016000    002000          JMPM*    GETR       GET FIELD ADDRESS(-1)
016001    100177
016002    005014          TAX
016003    015001          LDA      1,1        PICKUP BYTE ADDRESS
016004    004341          LSRA     1          MAKE WORD ADDRESS
016005    054032          STA      TMP1       SAVE
016006    010017          LDA      REND
016007    001004          JAN      ERROR      MORE ARGUMENTS
016010    016035
016011    002000          JMPM*    GETR       YES, GET DIRECTION INDICATOR
016012    100177
016013    005014          TAX
016014    015000          LDA      0,1        GET SIGN
016015    006150          ANAI     010000     ISOLATE +/- BIT (RIGHT/LEFT)
016016    010000
016017    054021          STA      TMP2       SAVE
016020    010017          LDA      REND
016021    001004          JAN      ERROR      MORE ARGUMENTS
016022    016035
016023    002000          JMPM*    GETR       YES, GET NUMBER OF PLACES
016024    100177
016025    005014          TAX
016026    015003          LDA      3,1        PICKUP D12-D15 OF BCD WORD
016027    006150          ANAI     0360       ISOLATE D14
016030    000360
016031    004344          LSRA     4          RIGHT JUSTIFY
016032    054007          STA      TMP3       SAVE

                    *     .
                    *     .
                    *     .
                    *   ETC.
                    *     .
                    *     .
                    *     .
016033    001000          JMP*     INT        EXIT TO RPG INTERPRETER
015034    100220
016035    000007    ERROR HLT      07         ERROR HALT
016036    001000          JMP*     INT
016037    100220
                    *
016040    000000    TMP1  DATA     0
016041    000000    TMP2  DATA     0
016042    000000    TMP3  DATA     0
                    *
000721              SHIFT ORG      0721
000721    151710          DATA     'SHIFT'    RUNTIME 'STAB' OVERLAY
000722    144706
000723    152240
000724    016000          DATA     SHIFT
                    *
000770              SHIFT ORG      0770
000770    151710          DATA     'SHIFT '   LOADER 'STAB' OVERLAY
000771    144706
000772    152240
000773    016000          DATA     SHIFT
                    *
     000000               END
```

Figure E-4. Sample Coding for DAS Subroutine

Master Operating System (MOS)

TABLE OF CONTENTS

SECTION 1

INTRODUCTION

SECTION 2

CONTROL DIRECTIVES

SECTION 3

INPUT/OUTPUT CONTROL PROGRAM

SECTION 4

DEBUGGING PROGRAM

SECTION 5

CONCORDANCE PROGRAM

SECTION 6

FILE EDITING PROGRAM

SECTION 7

SYSTEM MAINTENANCE PROGRAM

SECTION 8

SYSTEM PREPARATION PROGRAM

SECTION 9

LANGUAGE PROCESSORS

SECTION 10

SUPPORT LIBRARY

SECTION 11

MOS OPERATING PROCEDURES

SECTION 12

USER-CODED I/O DRIVERS

SECTION 13

STATUS AND ERROR MESSAGES

SECTION 14

MOS FORMATS

APPENDIX

SECTION 1 - INTRODUCTION

The **Master Operating System (MOS)** is a batch processing operating system for Varian computer systems. MOS operates on a wide range of hardware configurations. It is modular, thus facilitating expansion (e.g., new language processors, special user I/O drivers, etc.). MOS makes optimum use of memory by loading only those portions of the system (including I/O) required during execution. Features of MOS include:

- Minimum operator intervention required
- Single tape, drum, or disc as secondary storage device
- Extensive job control language (22 directives)
- Multisource input during loading
- Debugging aids
- File maintenance and editing programs
- Extensive status- and error-reporting

SYSTEM CONFIGURATION

The minimum MOS configuration requires the following hardware:

a. Varian computer

b. 4K memory

c. 33/35 ASR Teletype

d. One of the following:

   (1) Magnetic tape unit

   (2) Rotating memory unit on a buffer interlace controller (BIC)

MOS supports and is enhanced by the following hardware:

a. Card reader and/or punch

b. Line printer

c. High-speed paper tape reader and/or punch

d. Memory increment(s)

e. Hardware multiply/divide and extended addressing

MOS COMPONENTS

MOS is divided into resident and nonresident partitions. Figure I-1 shows their relationship.

RESIDENT PARTITION

This partition comprises the:

a. Resident monitor

b. Absolute loader

c. I/O assignment tables

d. System flags and parameters

e. Dump

NONRESIDENT PARTITIONS

This partition comprises the:

a. Control programs

b. Support programs

c. Language processors

Control Programs

The control programs are the:

a. Executive - job control processor and system control

b. System loader - linking and relocating loading of system and user programs

c. I/O control - dispatching of I/O requests and device driving

Support Programs

The support programs are:

a. Math and support library

b. Concordance program

c. Debugging program

d. File editing program

e. File maintenance program

f. System preparation program (operates in stand-alone mode)

Language Processors

The language processors are:

a. DAS MR assembler

b. FORTRAN IV compiler (requires an additional memory increment)

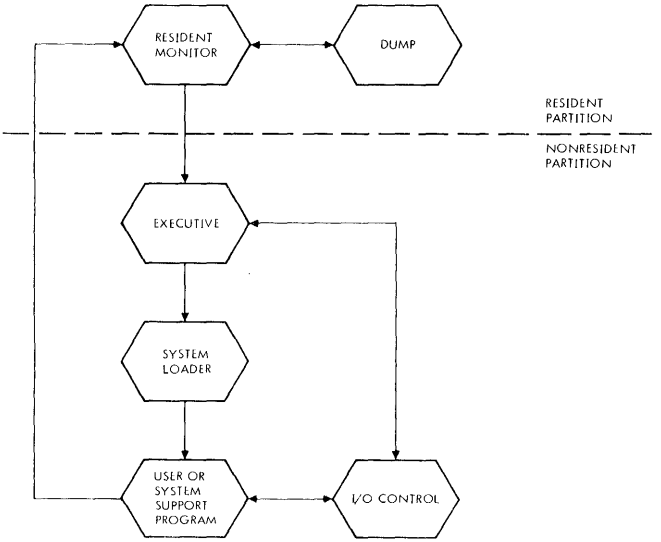


Figure 1-1. System Partitions and Flow

SECTION 2 - CONTROL DIRECTIVES

TYPES AND FORMAT

The MOS recognizes three types of control directives:

a. Executive control directives

b. I/O control directives

c. System loader directives

Executive and I/O control directives are executed while the executive is in memory. System loader directives cause the executive to be overlayed with the system program that is implemented by the directive. MOS has the following directives:

Executive	I/O	System Loader
JOB	COPYA	LOAD
ENDJOB	COPYB	ULOAD
ASSIGN	REW	ASSEMBLE
IOLIST	WEOF	FORTRAN
FORM	SREC	SMAIN
STACK	BREC	
EOF	SFILE	
DATE	FUNCTION	
COMMENT		

The general form of a directive is an alphanumeric record of up to 72 characters in the following format:

/name,p(1),p(2),p(3),...,p(n)

where

/ in the first character position of the record specifies that the record contains a control directive.

name is the name of the control directive comprising one to eight alphanumeric characters, and is terminated by a comma or blank.

p(1),... is a parameter string with individual parameters separated by commas.

The form and number of parameters varies with the directive. However, parameter strings cannot be longer than one record (72 characters). If it is, the directive is truncated.

The parameter string is terminated either by one period or by blanks from the last parameter to character position 72. Blanks within the parameter string are ignored.

Some control directives have an abbreviated form that is equivalent to the full form:

```
/C
/COMMENT
```

These forms can be used interchangeably.

EXECUTIVE CONTROL DIRECTIVES

```
JOB,title
```

This control directive starts a job. (A job is all tasks requested between a /JOB and an /ENDJOB directive.) The system is initialized by setting certain resident constants and logical unit assignments to their default values; a top-of-form function is sent to the list output.

One parameter (optional) is allowed. It is an alphanumeric string of up to eight characters that is stored in the resident monitor. It is an identification printed on every page of list output generated by the assembler or compiler. The identification is also incorporated into the actual object module. If the parameter has fewer than eight characters, the identification is left-justified and filled out with blanks. In the case of a /JOB parameter, the first blank terminates it, i.e., embedded blanks cause truncation.

The parameter can be accessed during execution by referencing the four-word area $TTL in the resident monitor. This can be done by making $TTL an external reference. The parameter is stored in ASCII, two characters per word.

```
ENDJOB
```

This control directive ends a job. The system is initialized by setting certain resident constants and logical unit assignments to their default values; a top-of-form function is sent to the list output. There are no parameters. Any text in the parameter field is ignored.

```
/ASSIGN,l(1)=r(1),l(2)=r(2),...,l(n)=r(n)
```

This control directive equates and assigns particular logical units to specific physical I/O devices. Execution of this directive decodes the parameter string and alters the logical unit table as specified by the parameter.

The parameters can be logical unit numbers, logical unit names, or physical unit names (figure 3-1). In each parameter pair (i.e., each l(n) = r(n)), the left parameter, l(n), is a logical unit number or name, and the right parameter, r(n), is a logical unit number or name or a physical device name.
In any case, the logical unit to the left of the equal sign is assigned to the unit/device to the right.

If r is a physical device, the l entry in the logical unit table is altered so that it points to the physical device driver specified by r. Thereafter, all I/O operations referencing l are directed to the physical device specified by r.

If r is a logical unit number or name, l is made equivalent to r and is assigned to the same physical device as r. However, if r is reassigned later to a new physical device, l no longer has an equivalent assignment.

As many parameter pairs as will fit in the control directive record can be specified on one /ASSIGN. Once a logical unit assignment is made, it remains in effect until changed by a new /ASSIGN, until the system is initialized by /JOB, /ENDJOB, or a bootstrap loading.

```
/IOLIST,p(1),p(2),...,p(n)
```

This control directive requests a listing of the current assignments of the individual logical units. The parameter string consists of logical unit numbers or names. As many logical units as will fit in the parameter field are allowed. The list is printed on LO and system output (SO, figure 3-1).

Example:

If the system file is currently assigned to drum unit 0 and the binary output to the high-speed paper tape punch, the directive

```
/IOLIST,SF,BO
```

prints the following on LO and SO:

```
SF = DR00
BO = PT00
```

If the parameter field is blank, all logical units and their assignments are printed, but only on LO.

```
/FORM, X
```

This control directive sets the value of the line-count word in the resident monitor to the value of parameter x. This word specifies, to operating system programs, the number of lines on the LO file before a top-of-form request is sent to the device. The parameter is a positive decimal integer from 5 to 9999. If the parameter is blank or less than 5, the value is set to the default value of 44.

```
/STACK
```

This control directive stacks binary object programs on the BO. Normally, the system rewinds the BO before each assembly or compilation. However, with /STACK in effect, the binary output of all tasks within a job are written on the BO file sequentially.

/STACK remains in effect until a /JOB or /ENDJOB is encountered. It is not set when the MOS is initialized after a bootstrap loading.

There are no parameters. The parameter field is ignored.

```
/EOF
```

This control directive instructs the executive to write an end-of-file record on the BO and GO (figure 3-1) files.

There are no parameters. The parameter field is ignored.

```
/DATE,xxxxxxxx
```

This control directive inputs the parameter into the operating system. The one parameter is an alphanumeric character string of up to eight characters (for example, month, day, and year). The first blank terminates it, i.e., embedded blanks cause truncation. The parameter is output on any list output generated by the assembler or compiler and other system support programs.

If the parameter field has fewer than eight characters, it is left-justified and filled out with blanks. Once entered, the parameter remains unchanged until another /DATE is encountered.

Access to the parameter during execution is by referencing the four-word area $DAT in the resident monitor. This can be done by making $DAT an external reference. The parameter is stored in ASCII, two characters per word.

```
/C
/COMMENT
```

This control directive annotates the list output. The MOS prints all 72 characters of this directive record, including /C or /COMMENT, on SO and LO. No other action is taken.

I/O CONTROL DIRECTIVES

```
/COPYA,l(1)=r(1),l(2)=r(2),...,l(n)=r(n)
```

This control directive copies ASCII-coded data files from one logical unit on another. The parameter string comprises logical unit pairs. Copying is record by record from the left (l) unit to the right (r) unit. Copying begins at the position of the l unit when the /COPYA is requested and continues until an end of file is encountered on the l unit. The r unit is not positioned before copying to it begins. After the copy is completed, neither unit is positioned nor is an end of file written on the r unit. Unit l is a logical unit number or name; unit r is a logical unit number or name.

```
/COPYB,l(1)=r(1),l(2)=r(2),...,l(n)=r(n)
```

This control directive is identical to /COPYA, except that copying is binary.

```
/REW,p(1),p(2),...,p(n)
```

This control directive rewinds the specified logical units. If a specified unit cannot be rewound, no action is taken for that unit.

```
/WEOF,p(1),p(2),...,p(n)
```

This control directive writes end-of-file records on the specified logical units. The format of the end-of-file record depends on the physical device to which the logical unit is currently assigned. If a physical device cannot accept an end of file, no action is taken for that logical unit.

```
/SREC,l(1),r(1),l(2),r(2),...,l(n),r(n)
```

This control directive skips physical records on the specified logical units. The parameter string consists of parameter pairs, each pair specifying a logical unit and a record count. /SREC spaces the logical unit forward the number of records designated.

The record count is a positive decimal integer from 0 to 9999. If the count is blank, or if the specified unit cannot skip records, no action is taken for that unit. If the count exceeds 9999, the left four digits are used.

```
/BREC,l(1),r(1),l(2),r(2),...,l(n),r(n)
```

This control directive is identical to /SREC, except that the records on the specified logical unit are skipped in reverse order (backspace).

```
/SFILE,l(1),r(1),l(2),r(2),...,l(n),r(n)
```

This control directive skips physical files on specified logical units. File-skipping occurs only in the forward direction. The parameter string consists of parameter pairs, each pair specifying a logical unit and a file count. /SFILE spaces the logical unit forward the number of physical files designated.

The file count is a positive decimal integer from 0 to 9999. If the count is blank, or if the specified unit cannot skip files, no action is taken for that unit. If the count exceeds 9999, the left four digits are used.

```
/FUNCTION
/FUNC,l(1),r(1),l(2),r(2),...,l(n),r(n)
```

This control directive performs special functions on specified logical units. The parameter string consists of parameter pairs, each pair specifying a logical unit and a special function code.

The special function code is a positive decimal integer from 0 to 9999. If the code is blank, no action is taken for that unit. If the code exceeds 9999, the left four digits are used.

The function performed depends on the physical device to which the logical unit is currently assigned. Definitions of the I/O functions of MOS are given in Section 3.

Note: Discs and drums must be positioned in the same manner as magnetic tapes with respect to the I/O control directives REW, SREC, BREC, and SFILE.

SYSTEM LOADER CONTROL DIRECTIVES

The system loader can load unconditionally from binary input (BI, figure 3-1) and/or selectively by program name from SF. It accepts only relocatable object text, including literal addressing and external program-linking. Upon successful completion, the system loader returns control to the resident monitor for program execution. When errors occur during loading, the process is aborted and control returned to the resident monitor, the executive is loaded and the error message posted on the LO and SO. Figures 2-1 and 2-2 show a map of memory during and after the loading process.

```
/LOAD
/L,p(1),p(2),...,p(n)
```

This control directive directs the executive to call the system loader and load one or more object modules from the BI file. The parameter string can specify the following tasks for the loader during loading, or for the resident monitor after the loaded program has been run:

PAUSE The loader pauses before each program is loaded from BI, allowing the loading of programs from more than one tape by stopping the computer during the changing of tapes.

HALT The resident monitor stops after all programs are loaded but before execution begins.

MAP When loading is complete, the loader outputs a map of all entry points, external names, and labeled data blocks (figure 2-3)

DUMP After the loaded program has been executed, the resident monitor dumps core on LO (figure 2-3).

The dump routine uses memory locations 0400 through 0477 and thus destroys the original contents of these locations.

DEBUG The loader loads the debugging program as part of the loading task.

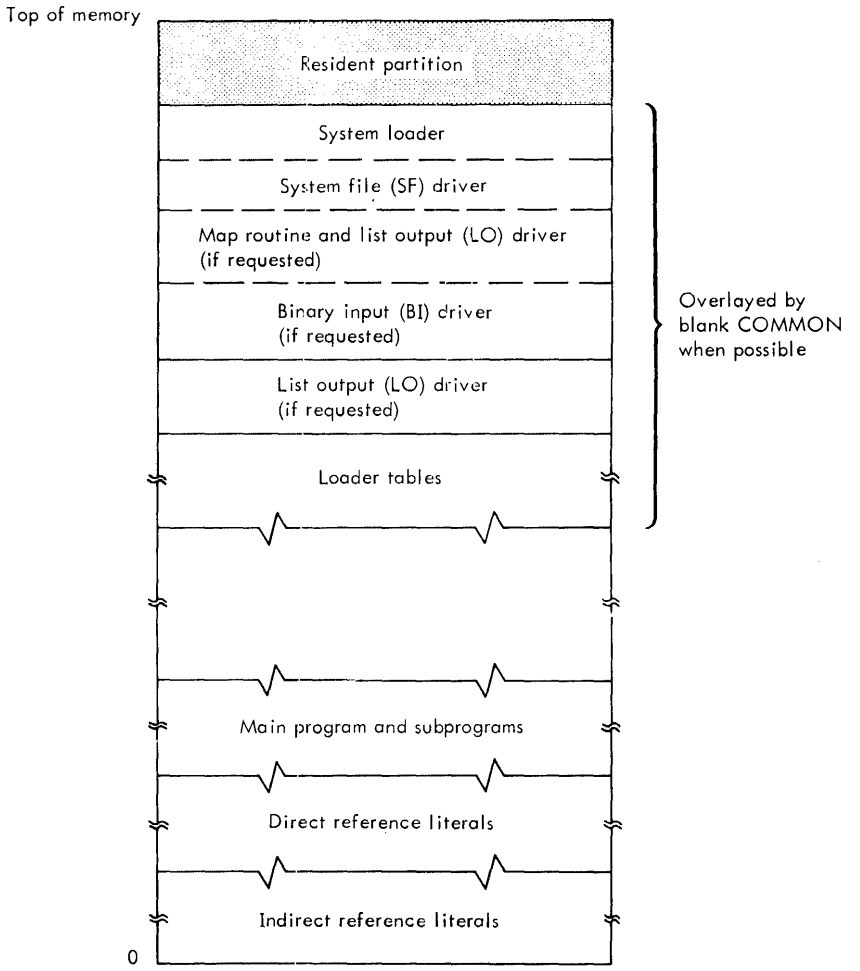


Figure 2-1. Loader Memory Map

THE FOLLOWING IS OUTPUT ON THE LO DEVICE WHEN A MAP REQUEST IS MADE WITH AN ASSEMBLY, COMPILATION, OR LOADING:

```
ssssss              aaaaa
.                   .
.                   .
ssssss/             rrrrr
.                   .
.                   .
($IAP)              vvvvv
($LIT)              vvvvv
($PED)              vvvvv
```

WHERE

ssssss IS A SYMBOLIC NAME, RIGHT-JUSTIFIED. THE CHARACTER / (SLASH) FOLLOWING ssssss INDICATES THAT THE ROUTINE WAS NOT LOADED.

aaaaa IS AN OCTAL ENTRY ADDRESS, RIGHT-JUSTIFIED, OF A LOADED PROGRAM OR COMMON BLOCK NAME.

rrrrr IS AN ADDRESS-REFERENCING OF AN UNLOADED PROGRAM OR COMMON BLOCK NAME.

vvvvv IS THE OCTAL VALUE OF:

- ($IAP) HIGHEST INDIRECT ADDRESS POOL LOCATION
- ($LIT) LOWEST DIRECT REFERENCE LITERAL POOL LOCATION
- ($PED) HIGHEST PROGRAM LOCATION STORED

Figure 2-2. Memory Map Format

```
000000  001000  035111  001000  034715  001000  034715  000350  103121
000010  000765  003712  105717  105720  105721  105776  000013  035656
000020  002013  001770  001771  001772  001773  002014  002015  004741
000030  001775  001776  002003  002004  002007  001375  001777  001300
000040  002000  001766  002005  002006  001374  000520  001276  001277
000050  001755  001757  001760  001761  001762  001763  120240  120240
000060  120240  120240  120240  120240  120240  120240  120240  120240
000070  120240  120240  120240  120240  120240  000610  010307  010306
000100  010674  010275  010276  102674  002674  000522  000523  000524
000110  000525  000526  001075  000516  000506  000507  000521  000520
000120  000510  000615  000615  001175  001176  001177  001200  000757
000130  001163  000625  000503  000504  000511  000505  000512  010305
000140  001162  001156  001155  000624  000762  001060  000623  000613
000150  000514  000617  000620  000621  000622  001165  010312  000744
000160  001164  005646  001157  000571  010302  010304  010273  010311
000170  000517  000561  000562  000564  010002  000513  001201  007345
000200  157777  007436  007350  007335  007315  007353  006674  000256
000210  000275  006563  177777  120200  000250  005175  006200  005722
000220  005563  005542  000100  000020  000040  005550  140711  154240
000230  146240  147640  151540  141540  123456  123456  123456  123456
000240  123456  123456  123456  123456  123456  123456  123456  123456
*
000400  130263  130264  130260  120240  120240  130663  130262  133260
000410  120240  130663  130266  133263  120240  130663  130262  133266
000420  130240  130663  131662  133263  120240  130662  130262  133260
000430  120240  130663  130266  133262  120240  130663  130262  133262
000440  120240  130663  131662  133262  120240  120240  120240  120240
000450  120240  120240  120240  120240  120240  120240  120240  120240
*
000470  120240  120240  120240  120240  003755  077777  000300  000004
000500  001003  000767  012760  023277  026404  000500  000010  000205
000510  012763  000222  016364  000500  000010  000205  140640  120240
000520  120240  120240  004732  140723  151705  146702  146305  004732
000530  140723  151711  143716  120240  003170  141322  142703  120240
000540  120240  003650  141640  120240  120240  120240  002647  141717
000550  146715  142716  152240  002647  141717  150331  140640  120240
000560  003760  141717  150331  141240  120240  003766  142301  152305
000570  120240  120240  002716  142716  142312  147702  000000  000000
000600  000000  000000  000000  000000  000000  000000  000000  000000
*
000620  000000  000000  000000  000000  000000  000000  120240  120240
000630  003645  143325  147303  152311  147716  003645  144717  146311
000640  151721  120240  003600  145317  141240  120240  120240  002730
000650  145240  120240  120240  120240  004254  146317  140704  120240
000660  120240  004254  151305  153640  120240  120240  003622  151706
000670  144714  142640  120240  003656  151715  140711  147240  120240
000700  005563  151722  142703  120240  120240  003653  151724  140703
000710  145240  120240  002744  152640  120240  120240  120240  004246
```

Note: An asterisk indicates that the succeeding line (or lines) has the same contents as the last printed line.

Figure 2-3. Dump Format

In addition to these task options, the parameter string can also specify relocation values to instruct the system loader where to relocate the program and data. Relocation value parameters are of the form:

xx = n

where

xx is the relocation bias name

n is the octal relocation value in the range 0 to 077777 entered as an octal number up to **five** digits (no leading zero is required).

The relocation bias names are:

RP Specifies the program relocation base
$0 \leq RP \leq$ core size

RI Specifies the indirect pointer relocation base
$0 \leq RI \leq 0777$

RL Specifies the literal relocation base
$0 \leq RL \leq 03777$

RC Specifies the COMMON relocation base
$0 \leq RC \leq$ core size

If these relocation biases are not specified, /LOAD sets them to the default values defined at system preparation time.

```
/ULOAD
/U,name,p(1),p(2),...,p(n)
```

This control directive directs the executive to call the system loader and load one or more programs from the SF file. Except for the name parameter, the parameters are identical to those of /LOAD.

The name parameter specifies the name of the program to be loaded from SF, and is the first parameter in the string. Object module names are generated by DAS MR or FORTRAN IV. Program names are alphanumeric character strings of one to eight characters. Six blanks is an illegal name. Parameters other than the name can appear in any order.

All error and bounds-checking of parameters is identical to that in /LOAD.

```
/ASSEMBLE
/A,p(1),p(2),...,p(n)
```

This control directive directs the executive to load the assembler. The parameter string specifies optional tasks for the assembler or executive to perform after the assembly is completed. These tasks are:

Parameter	Definition	Default Assignment
N	No source listing	Source listing
B	No binary object program output	Binary object program listing
MAP	Memory map on load-and-go	No memory map on load-and-go
L	Load-and-go after assembly	No load-and-go after assembly
M	No symbol table listing	Symbol table listing

If L (load-and-go) is specified, all the options and relocation parameters of /LOAD can be used in /ASSEMBLE. These loading parameters do not apply to the assembly, but to the load-and-go initiated after the assembly is completed.

```
/FORTRAN
/F,p(1),p(2),...p(n)
```

This control directive directs the executive to load the FORTRAN compiler. The parameter string specifies optional tasks that the compiler or executive is to perform. These tasks are:

Parameter	Definition	Default Assignment
N	No source listing	Source listing
B	No binary object program output	Binary object program listing
MAP	Memory map on load-and-go	No memory map on load-and-go
L	Load-and-go after compilation	No load-and-go after compilation
O	Octal listing of generated code	No octal listing of generated code
X	Conditional compilation	No conditional compilation
M	No symbol table listing	Symbol table listing

Parameter	Definition	Default Assignment
D	Generate two-word integer and logical numbers	Generate one-word integer and logical numbers

```
/SMAIN,p(1),p(2)
```

This control directive directs the executive to call the system loader and load the system maintenance program.

p(1), if present, is a physical unit name to which PI is assigned. p(2), if present, is a physical unit name to which PO is assigned. Neither the PI nor the PO can be assigned to dummy (DUM).

DECK PREPARATION

The batch processing facilities of MOS are envoked by control directives in combination with programs and data. These elements form the input job stream to MOS. The input job stream can come from various peripherals and be on various media. The following examples illustrate common job streams and deck preparation.

Example 1 - Card Input

Request a listing of all logical I/O assignments, enter the current date, set the LO line count to 50, and log a comment to the operator that reads: *MOUNT TAPE* DM72.

```
/JOB,EXAMPLE1
/IOLIST
/DATE,10/15/70
/FORM,50
/C,MOUNT TAPE DM72
/ENDJOB
```

Example 2 - Card Input

Compile a FORTRAN IV program with source listing, octal object listing, and load-and-go binary output.

```
/JOB,EXAMPLE2
/REW,GO
/FORTRAN,L,O
     .
(Source Deck)
     .
/EOF
/ENDJOB
```

Example 3 - Teletype Input

Assign the BI file to magnetic tape unit 2, load a program from BI and produce a map. After loading, halt before execution and produce a dump at program completion.

```
/JOB,EXAMPLE3
/ASSIGN,BI=MT01
/LOAD,MAP,HALT,DUMP
/ENDJOB
```

Example 4 - Card Input

Copy a card file from PI (processor input, figure 3-1) to scratch 1 (S1), write an end of file, rewind, and copy to the LO file. Then, copy scratch 1 (S1) to the second file of scratch 3 (S3), rewind both scratch tapes, and exit.

```
/JOB,EXAMPLE4
/COPYA,PI=S1
     .
(Card File)
     .
(End-of-File)
  .
/WEOF,S1
/REW,S1
/COPYA,S1=LO
/REW,S1,S3
/SFILE,S3,1
/COPYA,S1=S3
/WEOF,S3
/REW,S1,S3
/ENDJOB
```

SECTION 3 - INPUT/OUTPUT CONTROL PROGRAM

I O control is the generalized I/O subsystem under MOS for all users and requires only minimal understanding of the 620 computer hardware I/O operations. All I/O operations, both MOS and user-written, utilize I/O control. During program execution, only the required modules of I/O are loaded into memory.

Because of the standardized interfaces between I/O control and the user's program, and between I/O control and I/O subroutines, I/O peripherals can be changed without program reassembly. As new peripheral devices are added to a system, it is only necessary to program the required I/O driver (section 14).

Status and error messages are given in section 13.

LOGICAL AND PHYSICAL UNITS

MOS, through I/O control, allows access to I/O devices and files in terms of logical units with names and/or numbers rather than by actual physical references. The correspondence of a logical unit to a physical unit is made by /ASSIGN prior to program loading and execution.

MOS allows up to 64 logical units, with the first 14 being defined and used by MOS. Table 3-1 lists logical units defined by MOS; table 3-2, physical devices; and table 3-3, valid assignments.

Table 3-1. MOS I/O Units

Unit No.	Description	Unit Name	Function
1	System file	SF	The system file input logical unit. All processing, utility, and library programs are stored here. SF is a magnetic tape unit or drum or disc memory unit.
2	System input	SI	The system directive input logical unit. The operating system inputs all of its control directives from SI.
3	System output	SO	The system output logical unit. The operating system outputs all input control directives and outputs system operation messages on SO.
4	Processor input	PI	The language processor input logical unit. All operating system processors (assembler, compiler, etc.) input source statements from PI.
5	List output	LO	The system listing output logical unit. The operating system outputs all input control directives and any system operations messages on LO. All operating system processors (assembler, compiler, etc.) output listings on LO.

Table 3-1. MOS I/O Units *(continued)*

Unit No.	Description	Unit Name	Function
6	Binary input	BI	The system binary input logical unit. All operating system programs that input binary records input from BI (e.g., loaders).
7	Binary output	BO	The system binary output logical unit. All operating system processors (assembler, compiler, etc.) that output binary text records output on BO.
8	System scratch	SS	The system intermediate scratch logical unit. All operating system processors (assemblers, etc.) that use an intermediate scratch unit input from SS.
9	Load and Go	GO	The system assemble/compile and GO (load-and-go) logical unit. The assembler and compiler output on GO the same information as output on BO. When assemble/compile and GO is requested, GO references the system resident unit. Otherwise, it references a dummy I/O driver.
10	Processor	PO	The system processor output logical unit. All operating system processors (assembler, etc.) that use an intermediate scratch unit output on PO.

Table 3-1. MOS I/O Units *(continued)*

Unit No.	Description	Unit Name	Function
11	Scratch 1	S1	System scratch logical unit. For assignment as scratch records.
12	Scratch 2	S2	System scratch logical unit. For assignment as scratch records.
13	Scratch 3	S3	System scratch logical unit. For assignment as scratch records.
14	Scratch 4	S4	System scratch logical unit. For assignment as scratch records.
15-~~255~~ 64	User-assigned	Logical unit number	Can be assigned to any function.

Table 3-2. MOS Physical I/O Devices

System Name	Physical Device
——CPcu	Card punch
CRcu	Card reader
DKcu	Disc memory unit
DRcu	Drum memory unit
LPcu	Line printer
MTcu	Magnetic tape unit
PTcu	High-speed paper tape reader/punch
TPcu	Teletype paper tape punch
TRcu	Teletype paper tape reader
TYcu	Teletype printer
DUM	Dummy I/O

NOTES

1. cu represents controller/unit.

2.——DUM appears to the I/O control program to be a legitimate device, but in reality does nothing. DUM is used when an I/O output is not desired and for other special system purposes.

Table 3-3. Valid Logical Unit Assignments

SI	SO	PI	LO	BI	BO	GO	PO	S1,S2,S3,S4	SS
CR	TY	CR	CP	CR	CP	DR	DR	———DR	CR
DR		DR	DR	DR	DR	MT	MT	———MT	DR
MT		MT	LP	MT	MT			———DUM	MT
PT		PT	MT	PT	PT			———CR	PT
TR		TR	PT	TR	TP			———PT	TR
TY		TY	TR	DUM	DUM			———TR	TY
		DUM	TY					———TY	DUM
			DUM					———LP	

* For disc versions of MOS, every DRxx unit name is replaced by DKxx.

I/O CALLS

During program execution, I/O control facilities are accessed through calls specifically defined by the DAS MR assembler. When called, I/O control normally transfers one record at a time between the computer memory and the I/O device. An I/O request to I/O control specifies:

a. The logical unit number

b. The type of I/O function to be performed

c. The number of memory words to be transmitted, or a count for skip operations

d. The data location

These parameters are combined into 14 I/O calls recognized by the DAS MR assembler (table 3-4).

The following abbreviations are used to describe the calls:

Abbreviation	Definition
DST	Device specification table
fc	Function code
IOCS	I/O control entry point
loc	Data address
lun	Logical unit number
n	Number of words, records, or files to be skipped
wc	Word count

NOTE

The A, B, and X registers and the overflow indicator are assumed volatile and are destroyed during all I/O calls.

Table 3-4 Summary of I/O Calls

Mnemonic	Definition	Function Code
FUNC	Perform function	016
RALF	Read alphanumeric record	001
RBCD	Read BCD record	041
RBIN	Read binary record	101
REW	Rewind	004
SKFF	Skip files forward	005
SKFR	Skip files reverse	205
SKRF	Skip records forward	006
SKRR	Skip records reverse	206
STAT	Request status	000
WALF	Write alphanumeric record	002
WBCD	Write BCD record	042
WBIN	Write binary record	102
WEOF	Write end of file	003

The general format of all I/O calls is:

Label Field	Operation Field	Variable Field
Symbol (optional)	Mnemonic	1, 2, 3, or 5 parameters or expressions separated by commas

The general format of the expansion of all I/O calls is:

Label Field	Operation Field	Variable Field
Symbol (optional)	JMPM	IOCS
	DATA	fc + parameter 1
	.	parameter 2
	.	
	.	
	DATA	parameter n

where (fc + parameter 1) is:

Bit Position

```
17* 16 15    14 13 12    11 10 9    8 7 6    5 4 3    2 1 0
       n     n  n  n     n  n  n    n n n    n n n    n n n
       ---------------------------- ====================
              Function Code      +          lun (p1)
```

where n is the octal value of the indicated bit groups.

* For 18-bit computers (622).

In addition to the expansion of each call, the assembler generates two other directives at the beginning of every program that uses I/O calls:

```
EXT       IOCS
ION       lun 1,...,lun n
```

The EXT declares IOCS an external reference, while ION does the same for the necessary drivers. Both allow the loader to link the user and I/O control at load time.

The following sections give a detailed discussion of each I/O call defined within MOS.

READ BINARY RECORD

Call:

```
RBIN    lun,wc,loc
```

Expansion:

```
JMPM    IOCS
DATA    040400+lun
DATA    wc
DATA    loc
```

This call inputs wc words from the I/O device to consecutive memory addresses beginning at loc. The function is performed in the mode requested, if possible. Otherwise, the mode of the device is used. If the input record contains more than wc words, only wc words are stored in memory, and the remainder, ignored. If the input record contains less than wc words, they are input. The number of words is placed in word 0 of the DST. The I O request is ignored if wc is not greater than zero.

READ ALPHANUMERIC RECORD

Call:

```
RALF    lun,wc,loc
```

Expansion:

```
JMPM    IOCS
DATA    000400+lun
DATA    wc
DATA    loc
```

This call inputs wc words from the I/O device to consecutive memory addresses beginning at loc. The function is performed in the mode requested, if possible. Otherwise, the mode of the device is used. If the input record contains more than wc words, only wc words are stored in memory, and the remainder, ignored. If the input record contains less than wc words, they are input. The number of words is placed in word 0 of the DST. The I/O request is ignored if wc is not greater than zero.

READ BCD RECORD

Call:

```
RBCD    lun,wc,loc
```

Expansion:

```
JMPM    IOCS
DATA    020400+lun
```

```
DATA        wc
DATA        loc
```

This function inputs wc words from the I/O device to consecutive memory addresses beginning at loc. The function is performed in the mode requested, if possible. Otherwise, the mode of the device is used. If the input record contains more than wc words, only wc words are stored in memory, and the remainder, ignored. If the input record contains less than wc words, they are input. The number of words input is placed in word 0 of the DST. The I/O request is ignored if wc is not greater than zero.

WRITE BINARY RECORD

Call:

```
WBIN        lun,wc,loc
```

Expansion:

```
JMPM        IOCS
DATA        041000+lun
DATA        wc
DATA        loc
```

This function outputs wc words to the I/O device from consecutive memory addresses beginning at loc. The function is performed in the mode requested, if possible. Otherwise, the mode of the device is used. If output is specified with less than wc words, they are output. If the configuration permits wc or more words, wc words are output. The number of words output is placed in word 0 of the DST. The I/O request is ignored if wc is not greater than zero.

WRITE ALPHANUMERIC RECORD

Call:

```
WALF        lun,wc,loc
```

Expansion:

```
JMPM        IOCS
DATA        001000+lun
DATA        wc
DATA        loc
```

This function outputs wc words to the I/O device from consecutive memory addresses beginning at loc. The function is performed in the mode requested, if possible. Otherwise, the mode of the device is used. If output is specified with less than wc words, they are output. If the configuration permits wc or more words, wc words are output. The number

of words output is placed in word 0 of the DST. The I/O request is ignored if wc is not greater than zero.

Efficient choice of a record length depends upon the peripheral device: for example, the more efficient use of a disc file would be with record lengths in a multiple of sixty words.

WRITE BCD RECORD

Call:

```
WBCD    lun,wc,loc
```

Expansion:

```
JMPM    IOCS
DATA    021000+lun
DATA    wc
DATA    loc
```

This function outputs wc words to the I/O device from consecutive memory addresses beginning at loc. The function is performed in the mode requested, if possible. Otherwise, the mode of the device is used. If output is specified with less than wc words, they are output. If the configuration permits wc or more words, wc words are output. The number of words output is placed in word 0 of the DST. The request is ignored if wc is not greater than zero.

Note

For high-speed paper-tape reading and writing there is a driver for unformatted tapes as well as for formatted tapes. The high-speed paper-tape driver for unformatted tape makes no distinction between WBIN and WALF. On input this driver will read one record which is defined by the wc parameter in the program if wc is less than 60, or otherwise 60 words will be read. Each record read is checked for leading blank characters. If the first half record contains all blank characters, the blanks are interpreted as an end-of-file. On output, the driver for unformatted tapes will write two frames per word to paper tape for wc words. The DST and paper-tape driver for unformatted tapes are combined as one routine labeled $OP, which corresponds to the peripheral mnemonic PT10.

WRITE END OF FILE

Call:

```
WEOF    lun
```

Expansion:

```
JMPM    IOCS
DATA    001400+lun
```

This function outputs an end-of-file character to the peripheral device. No data are transmitted. Subsequent I/O status requests produce an end-of-file return.

REWIND

Call:

```
REW       lun
```

Expansion:

```
JMPM      IOCS
DATA      002000+lun
```

This function issues a rewind command to the I/O device. No data are transmitted. Subsequent I/O status requests produce a beginning-of-device return.

SKIP RECORDS FORWARD

Call:

```
SKRF      lun,n
```

Expansion:

```
JMPM      IOCS
DATA      003000+lun
DATA      n
```

This function skips n records in the forward direction on the I/O device. No data are transmitted. If an end of device or end of file is detected before the requested number of records are skipped, skipping terminates and the count of records remaining to be skipped is placed in word 0 of the DST. Subsequent I/O status requests produce an end-of-device or end-of-file return, respectively. The request is ignored if n is not greater than zero.

SKIP RECORDS REVERSE

Call:

```
SKRR      lun,n
```

Expansion:

```
JMPM      IOCS
DATA      0103000+lun
DATA      n
```

This function skips n records in the reverse direction on the I/O device. No data are transmitted. If a beginning of device or an end of file is detected before the requested number of records are skipped, skipping terminates and the count of records remaining to be skipped is placed in word 0 of the DST. Subsequent I/O status requests produce a beginning-of-device or end-of-file return, respectively. The request is ignored if n is not greater than zero.

SKIP FILES FORWARD

Call:

```
SKFF    lun,n
```

Expansion:

```
JMPM    IOCS
DATA    002400+lun
DATA    n
```

This function skips n file marks in the forward direction on the I/O device. No data are transmitted. Subsequent I/O status requests produce an end-of-file return. If an end of device is detected before the requested number of files are skipped, skipping terminates and the count of any remaining files to be skipped is placed in word 0 of the DST. Subsequent I/O status requests produce an end-of-device return. The request is ignored if n is not greater than zero.

SKIP FILES REVERSE

Call:

```
SKFR    lun,n
```

Expansion:

```
JMPM    IOCS
DATA    0102400+lun
DATA    n
```

This function skips n file marks in the reverse direction on the I/O device. No data are transmitted. Subsequent I/O status requests produce an end-of-file return. If an end of device is detected before the requested number of files are skipped, skipping terminates and the count of any remaining files to be skipped is placed in word 0 of the DST. Subsequent I/O status requests produce a beginning-of-device return. The request is ignored if n is not greater than zero.

PERFORM FUNCTION

Call:

```
FUNC    lun,n
```

Expansion:

```
JMPM    IOCS
DATA    003400+lun
DATA    n
```

This function commands one of the following special functions peculiar to the specified I/O device:

Peripheral	Function
Teletype keyboard (TYcu)	Spaces paper five lines (n ignored)
Teletype paper tape punch (TPcu)	Punches about 24 inches of blank leader (n ignored)
High-speed paper tape punch (PTcu)	Punches about 24 inches of blank leader (n ignored)
Card punch (CPcu)	Ejects one blank card (n ignored)
Line printer (LPcu)	Slews paper at a rate other than one line per print line (n is the slew character). The page position for each of the eight possible counts in a function call is listed below:

Value of n	Channel on Format Tape	Position on Page
0	0	Top of form
1	1	Reserved for user
2	2	Reserved for user
3	3	Reserved for user
4	4	Reserved for user
5	5	Reserved for user
6	6	Reserved for user
7	7	Reserved for user

REQUEST STATUS

Call:

```
STAT    lun,err,eof,beod,busy
```

Expansion:

```
JMPM    IOCS
DATA    0+lun
DATA    ERR
DATA    EOF
DATA    BEOD
DATA    BUSY
```

This function examines the I/O driver of a logical unit to determine its status and then specifies return addresses to be used depending on this status. Return addresses can specify indirect addressing. On return, the X register points to word 0 of the DST of the I/O driver for this lun. If this lun has no I/O driver, the X register is cleared.

The various exit terms are:

ERR	I/O error on last transfer
EOF	End of file on last transfer
BEOD	Beginning/end of device on last transfer
BUSY	Device busy

I/O PROGRAMMING EXAMPLES

Example 1

Rewind tape unit 3 (lun = 15) and read one file consisting of 100 records of 60 binary words each. After each record is read, print it on the line printer (lun = 5). Go to the top of the next form upon completion and rewind tape unit 3. Halt on an end of file or I/O error. Exit to the resident monitor on normal completion.

Label Field	Operation Field	Variable Field
A	REW	15
	LDAI	-100
B	RALF	15,60,BUF
C	STAT	15,X,X,X,C
	WALF	5,60,BUF
D	STAT	5,X,X,X,D
	IAR	
	JAN	B
	FUNC	5,1
E	STAT	5,X,X,X,E
	REW	15
F	STAT	15,X,X,G,F
BUF	BSS	60
X	HLT	0
	EXT	EXIT
G	CALL	EXIT
	END	A

Example 2

Read a card from the card reader (lun = 6) until an end of file is detected and write it on a drum file (lun = 20). List all I/O errors on the Teletype, but do not terminate the operation.

Label Field	Operation Field	Variable Field
A	RBIN	6,60,BUF
B	STAT	6,F,D,F,B
	WBIN	20,60,BUF
E	STAT	20,F,F,F,E
	JMP	A
F	WALF	3,3,H
J	STAT	3,A,A,A,J
	JMP	A
	EXT	EXIT
D	CALL	EXIT
BUF	BSS	60
H	DATA	'IO ERR'
	END	A

Example 3

Read a disc file (lun = 17) consisting of 40-word records until and end of file is detected. Search each record for a zero word and keep a count of them. At end of file, punch a binary card on lun = 7 with the count of zero words in the first card word. Ignore I/O errors.

Label Field	Operation Field	Variable Field
A	TZA	
	STA	H
B	RBIN	17,40,BUF
D	STAT	17,C,G,C,D
C	LDXI	BUF
E	LDA	0,X
	XAZ	F
	INCR	045
	SUBI	BUF + 40
	JAP	B
	JMP	E
G	WBIN	7,1,H
I	STAT	7,J,J,J,I
	EXT	EXIT
J	CALL	EXIT
F	INR	H
H	DATA	0
BUF	BSS	40
	END	A

SECTION 4 - DEBUGGING PROGRAM

The MOS debugging program aids the programmer in finding and correcting program errors. Its commands examine and/or change the program, in addition to running part or all of a program.

Whenever the DEBUG option is specified on /LOAD or /ULOAD (section 2), the debugging program is loaded with the user's program. When loading is complete, control is transferred to the debugging program.

Status and error messages are given in section 13.

DIALOG

The debugging program is an interactive component within MOS used via SI. Upon entry, if SI = TY00, it types:

```
C/R**
```

To communicate with the debugging program, use the command language with the following syntax:

a. General form:
   instruction parameter(1),parameter(2),...,parameter(n)

b. Instructions can have up to 72 characters. Characters beyond the 72nd are ignored.

c. Continuation lines begin with a comma (,).

d. Invalid instructions cause the reply **WHAT?? followed** by ** if SI = TY00.

e. All numbers are octal.

f. **Negative (two's complement) numbers are preceded by a minus (–) sign.**

g. Parameters are separated by commas.

h. Blanks between parameters are ignored, but other blanks may cause errors.

PSEUDOREGISTERS

Because the debugging program uses the A, B, and X registers and the overflow indicator, pseudoregisters are defined to guarantee the integrity of the physical registers during the debugging process. The debugging program loads the pseudoregisters into the physical registers prior to transferring control to the user's program. With a breakpoint (trap) set, the contents of physical registers are saved in the pseudoregisters when the breakpoint is reached. When the program starts from the breakpoint, the physical registers are restored to the saved values. The pseudo-overflow indicator contains zero or nonzero, corresponding to overflow reset or set, respectively.

INSTRUCTION LANGUAGE

The following is a list of instructions accepted by the debugging program, where aaaaaa denotes any signed or unsigned 16-bit (18-bit for 622 computers) octal number. The debugging program makes no check of addresses against the actual memory size. Use care in specifying addresses.

DISPLAY AND ALTER INSTRUCTIONS

Instruction	Description
aaaaaa	Display the contents of memory at the given memory address on LO.
aaaaa(1),aaaaaa(2)	Display the contents of memory at aaaaa0(1) through aaaaa7(2) inclusive on LO. An asterisk indicates that the succeeding line (or lines) has the same contents as the last printed line.
A	Display the contents of the pseudo-A register on LO.
Aaaaaaa	Change the contents of the pseudo-A register to aaaaaa.
B	Display the contents of the pseudo-B register on LO.
Baaaaaa	Change the contents of the pseudo-B register to aaaaaa.
X	Display the contents of the pseudo-X register on LO.
Xaaaaaa	Change the contents of the pseudo-X register to aaaaaa.

Instruction	Description
O	Display the contents of the pseudo-overflow indicator on LO.
Oaaaaaa	Change the contents of the pseudo-overflow indicator.
Caaaaaa,v(1),v(2),..., v(x)	Change the contents of memory addresses aaaaaa and following to the values v(1) to v(x), where x = 1 through 16 and v has the same range as aaaaaa.
Iaaaaaa(1),aaaaaa(2),v	Initialize the contents of memory addresses aaaaaa(1) through aaaaaa(2) to the value v, where v has the same range as aaaaaa.
Saaaaaa(1),aaaaaa(2),v	Search through memory addresses aaaaaa(1) to aaaaaa(2) for value v and log all addresses containing that value on LO.
Saaaaaa(1),aaaaaa(2),v,m	Search through memory addresses aaaaaa(1) to aaaaaa(2) for the value v. The contents of each memory address are ANDed with the mask specified by m prior to comparison with v and log all addresses containing that value on LO.

I/O INSTRUCTIONS

Instruction	Description
ASSIGN,I(1) = r(1), I(2) = r(2),..., I(x) = r(x)	Assign logical units I(1) through I(x) to physical devices r(1) through r(x), respectively, I(1) through I(x) are decimal numbers. Re-assignments can be made only to physical devices whose drivers were loaded with DEBUG.
IOLIST,I(1),I(2),..., I(x)	List current logical unit assignments (section 2).
R	Read a program in binary record format (see appendix C) from logical unit assigned to BI into memory. When reading is complete, print the starting address, ending address, execution address, and program name on the LO. **Caution: Only programs that have been punched by DEBUG can be read by DEBUG.**

Instruction	Description
T	Terminate the current printout after the next octal number. This can be used for discontinuing search, memory display, or I/O list output.
Waaaaaa(1),aaaaaa(2)	Write memory in binary record format (see appendix C) on the physical unit assigned to BO from addresses aaaaaa(1) through aaaaaa(2).
Waaaaaa(1), aaaaaa(2) aaaaaa(3)	**Write memory in binary record on the physical unit assigned to BO from addresses aaaaaa(1) through aaaaaa(2) and set the execution address to aaaaaa(3).**
Waaaaaa(1),aaaaaa(2), aaaaaa(3),name	Write memory in binary record on the physical unit assigned to BO from addresses aaaaaa(1) through aaaaaa(2). Set the execution address to aaaaaa(3) and name the program, where name represents any legal MOS label.

CONTROL INSTRUCTIONS

Instruction	Description
Gaaaaaa	Load the registers from the pseudoregisters and transfer control to address aaaaaa.
Taaaaaa(1),aaaaaa(2)	Set a trap to the debugging program in memory address aaaaaa(1) and transfer control to address aaaaaa(2). Load the registers from the pseudo-registers prior to transferring. Upon reaching the breakpoint (trap), save and type the contents of the registers on LO.

EXAMPLES OF DEBUGGING

In the following examples, symbols in bold type indicate user-entered information. Other symbols represent the output of the debugging program.

Display and Alter Instructions

****A**	Display the contents of the pseudo-A register.
(077553)	Contents of the pseudo-A register.
****A005572**	Change the contents of A to 005572.
****B**	Display the contents of the pseudo-B register.
(177750)	Contents of the pseudo-B register.
****B-1**	Change the contents of B to minus one (0177777) (0777777 on 622 computers)
****X**	Display the contents of the pseudo-X register.
(000021)	Contents of the pseudo-X register.
****X12**	Change the contents of X to 000012.
****O**	Display the contents of the pseudo-overflow indicator.
(000000)	Contents of the pseudo-overflow indicator.
****O-1**	Set the pseudo-overflow indicator.
******26001	Display the contents of memory address 026001.
(170522)	Contents of memory address 026001.
****1002,1045**	Display the contents of memory addresses 001002 through 001045.

```
001000  177776  100001  010101  151501  025252  000000  000101  015432
001010  144456  052345  177777  101010  111101  063063  033333  177777
          .       .       .       .       .       .
          .       .       .       .       .       .
          .       .       .       .       .       .
001040  177776  177776  177775  020205  123456  000000  035353  077756
```

****C26000,1000,2500** ****,77777,375**	Change the contents of addresses 026000 through 026003 to 001000, 002500, 077777, and 000375, respectively.
****I1000,1100,125252**	Initialize addresses 001000 through 001100 to 0125252 (each word from 001000 through

001100 contains 0125252).

****S5000,6000,100000** — Search addresses 005000 through 006000 for 0100000 and print each address where this value is found.

005100
005302
005701

In this example, 0100000 was found in addresses 005100, 005302, and 005701

****S6300,10000,136000, 177000** — Search addresses 006300 through 010000 for the value 0136 in bits 9 through 15 of each word and print each such address and its full contents. The last parameter, 017700, is used as a mask.

006320
006700
006701

In this example, three locations met the comparison criteria.

(136111)
(136111)
(136000)

I/O Instructions

****ASSIGN,12 = MT01** — Assign logical unit 12 to device MT01 (magnetic tape unit 01). The unit number is decimal.

****IOLIST** — List the assignment of logical units to physical devices.

```
01    Blank
02    CR00
03    TY00
04    Blank
.     .
.     .
.     .
.     .
15    MT03
```

In this example, LUN 2 is the card reader, LUN 3, is the Teletype, and LUN 15 is magnetic tape unit 03. LUN 1 and LUN 4 were not loaded with DEBUG.

****W17000,17556,17000, TEST** — Write the program named TEST on the BO in absolute format (TEST resides in addresses 017000 through 017556)and set execution starting at 017000.

****W14000,14100,TEST**
****W14700,15334** — Write the program named TEST on the BO in absolute format (TEST resides in addresses

****W15734,16040,14740**	014000 through 014100, 014700 through 015334, and 015734 through 016040). Set execution starting at 014740.
****R**	Read an absolute format program from the BI into memory.
017000 017556 017777 TEST	When reading is completed, print the starting address, ending address, execution address, and program name.

Control Instructions

****G5013**	Transfer the contents of the pseudo-A, -B, and -X registers and the pseudo-overflow indicator and execute at address 005013.
****T26000,21000**	Load the physical registers from the pseudoregisters and transfer to address 021000. If and when breakpoint (trap) address 026000 is reached, save and type.
026000 (001004) 001111 150000 000025 000001	The breakpoint (trap) address 026000, its contents 001004, and the contents of the A, B, and X registers and the overflow indicator.
****T15000**	Load the registers from the pseudoregisters and continue program execution from the prior breakpoint (in the above example, execution would continue from 026000) and use 015000 as the new breakpoint. If address 015000 is reached, save and type.
015000 (054002) 000001 111000 010101 000000	The breakpoint (trap) address 015000, its contents 054002, and the contents of the A, B, and X registers and the overflow indicator.

SECTION 5 - CONCORDANCE PROGRAM

The concordance program is an MOS support program that analyzes the symbols of a DAS assembler program. The analysis consists of a printout showing where symbols are defined and referenced. The analysis can be performed on any source program in DAS assembler languages.

Upon being loaded by /ULOAD,CONC (Section 2), the concordance program inputs source programs from the system scratch (SS) logical unit (lun = 8). Any of the following terminates the input:

a. END card

b. MOS control directive

c. End-of-file or end-of-device status received

d. Available memory is full

When any one of these conditions arises, a concordance list is output on the list output (LO) logical unit. If the list was output for one of the first three reasons, the concordance program exits to the resident monitor. If the list was output because no more memory space is available, the concordance program clears the concordance table and continues with another concordance. This continues until one of the other terminating conditions is met.

The concordance program does not position SS except when it is MT00. In this case, one file is skipped and the concordance starts with the second file.

Figure 5-1 is a concordance listing output. A title line at the top of each page consists of the page number, program name (blank if none) and the date (blank if not input with /DATE). Following the title line is the concordance.

```
PAGE    1  RSCR2LPC  11/06/70

  8  $BIT       52    86   121   123   125   127   135   137
  0  $JCW       17    18
  0  $LAX      188   189
  0  $LBF      157   158   159
  0  $ROC      116   117
193  $XEQ      192
159  A$BF       43    46    68    73    96   100
 10  BASE       15
191  BLK        40   143
158  BUFA       39    49    72    74   101   151   154
 92  CHAR       87
104  D100      178
 70  D60        44
 24  DMP        13    14
 38  DMPG      112
143  DPR       146
 27  DSCR      142
114  DSDC      109
185  DSDE      118
122  DSDL       26   136   138   148
 50  DSFB       71    99
140  DWHK      134
149  DWIT      111   131   144   155
148  JMP       124
  6  LPT        47   161
186  NODMP      12    21
160  O10        20
173  O200
163  PAGE       25   165   166   176   179   185
 16  RSCR2     194
168  SENS       79    82    88    93   102   164   169   170   175   177
                182   183
156  TEMP       38    78   107   115   150   153
172  UUU       162
184  VVV       171
108  WWW        80
 81  XXX        46
 63  YYY        58
 59  ZZZ        53
```

Figure 5-1. Sample Concordance Listing

Beginning with the first character position, the format for a concordance line is:

a. Four positions to display the decimal line number where the symbol is defined. The line number is right-justified and left-blank-filled.

b. Two blanks.

c. Six positions to display the symbol, in sort sequence, left-justified and right-blank-filled.

d. Up to ten reference line numbers. Each reference line number is in sort sequence, preceded by two blanks and four character positions to display the decimal line number, right-justified and left-blank-filled.

The maximum line is 72 characters. Continuation lines contain only the reference line numbers.

Status and error messages are given in Section 13.

SECTION 6 - FILE EDITING PROGRAM

The file editing program is a support program operating under MOS that is a simple and powerful tool for editing files generated under MOS (e.g., source programs, text, etc.).

PROGRAM AND DIRECTIVES

The user supplies the file editing program with control directives and files, and the program produces new and/or modified files and an audit trail of all transactions.

Status and error messages are given in Section 13.

The file editing program accepts single-reel files and multifile reels of input. It processes input in control directive and source file format, and outputs source file format and a printed audit listing.

Control directives are input through the SI.

On being loaded by /ULOAD,EDITOR,RP = 0500,RI = 0377 (Section 2), the file editing program types

```
BEGIN EDITOR
```

on the LO and inputs control directives to obtain its instructions from the user. The following conventions are used in describing and coding file editing directives:

a. General form:

   n,p(1),p(2),...,p(x)

   where n is the directive name, and p(1),p(2),...,p(x) is a parameter string with individual parameters separated by commas.

b. Directives begin in the first character position of the record and can consist of up to 72 characters.

c. All fields are interpreted as fixed-format data and must be the exact size as indicated below.

d. Parameters shown in upper-case letters appear on the control records exactly as shown.

e. Parameters in italics are optional.

f. Parameters in lower-case letters are to be replaced by user-defined character strings according to the following:

filename	Name of file: any **eight** characters
author	Author of file: any **12** characters
aaa	Alphabetic portion of sequence number: any **three** alphabetic characters
nnnnn	Numeric portion of sequence number: any **five** numeric digits ending in zero
location	Either the internal sequence number (in the form aaannnnn) or the external line number (in the form nnnnn) identifying a record
nnn	Number of records: any **three** numeric digits
l	The letter l at the end of a parameter string indicates that location is an internal sequence number.

DIRECTIVES

LIST,*xxx*

LIST specifies list output during editing. If omitted, no list output is provided. Its presence, with or without a parameter, causes a file audit and outputs catalog to be printed (figure 6-1). The parameter xxx, if used, is:

a. *CHG* - list changes only

b. *ALL* - list complete file(s)

LIST can be repeated for different options on different files. It immediately precedes a FILE, COPY, or EDIT.

When an audit listing is output, it is produced on the LO. Three levels of reporting are provided. All include a heading identifying the report as an output and giving the run date (Figure 6-2).

```
FILE EDIT RUN                OUTPUT CATALOG    10/14/70        PAGE0003

        FILE NAME            # OF RECORDS          CHANGED

        NUM1                      0008                0287

        NUM2                      0013                0287

**
```

Figure 6-1. Output Catalog Format

FILE EDIT RUN FILE AUDIT 10/14/70 PAGE0002

FILENAME IS NUM2 AUTHOR IS JONES CREATION DATE IS 10/14/70

CONTROL RECORD OR DATA RECORD IMAGE				LINE#	DATE	CODE
C	WBIN	BO,10,D	VDM00000	00001	0287	ADD
	STAT	BO,ERR,EOF,BEOD,*-6	VDM00010	00002	0287	ADD
	EXT	E	VDM00020	00003	0287	ADD
	JMP	E	VDM00030	00004	0287	ADD
D	BSS	10	VDM00040	00005	0287	ADD
BO	EQU	7	VDM00050	00006	0287	ADD
ERR	INCR	01	VDM00060	00007	0287	ADD
EOF	INCR	02	VDM00070	00008	0287	ADD
BEOD	INCR	04	VDM00080	00009	0287	ADD
	EXT	F	VDM00090	00010	0287	ADD
	JMP	F	VDM00100	00011	0287	ADD
	END	C	VDM00110	00012	0287	ADD
/ENDJOB			VDM00120	00013	0287	ADD

NOTES

1. CONTROL RECORDS ARE PRECEDED AND FOLLOWED BY TWO BLANK LINES.
2. GROUPS OF ADDITIONS OR DELETIONS ARE PRECEDED AND FOLLOWED BY ONE BLANK LINE.
3. EACH NEW FILE BEGINS ON A NEW PAGE.
4. LITING OUTPUT SETUP USING TELETYPE AS LO.

Figure 6-2. Audit Listing Format

The output catalog report is a summary of the contents of the source library showing the file names and the number of records each contains. The date of the latest change to each file (excluding resequencing only) is shown.

The file audit report is available in two levels for each file on the source library. All records in the file can be shown, or the report can be limited to those records having changes. Each record listed is accompanied by a program-generated line number and an action code (ADD) if the record has been added. All control records are also listed, and have four asterisks in the action code field.

FILE,filename,author,*SEQ,aaannnnn*

FILE creates a file from the data records following it. The file is formatted as a standard MOS source file (section 3). The optional *SEQ* parameter assigns internal sequence numbers in positions 73 through 80 of each record, beginning with the value specified in *aaannnnn* and incrementing by ten for each record.

FILE is followed immediately by the data records comprising the file.

An end of file following the data records indicates the end of the data to be processed with FILE. This can be a 2-7-8-9 punch on cards; a BELL character on the Teletype; or an EOF mark on paper tape, magnetic tape, drum, or disc.

COPY,filename,*filename,SEQ,aaannnnn*

COPY copies a file or files from the input device to the output device. If a second filename is specified, all files from the one specified in the first filename through that of the second filename are copied. If *SEQ* and *aaannnnn* are present, they apply only to the first file being copied and cause the program to assign internal sequence numbers in positions 73 through 80 of each record being copied. Sequence numbers start with *aaannnnn* and increment by ten for each record.

EDIT,filename,*SEQ,aaannnnn*

EDIT copies a file from the input device to the output device with the modifications specified by the ADD, DEL, and RPL that immediately follow EDIT. If SEQ and aaannnnn are present, they apply only to the first file being copied and cause the program to assign internal sequence number in positions 73 through 80 of each record being copied. Sequence numbers start with aaannnnn and increment by ten for each record.

ADD,nnn,location,*I*

ADD adds the following nnn data records to the file being processed immediately after the record specified by location. If location represents an internal sequence number rather than the line number of the previous audit list, add I.

This control record is valid only after an EDIT and before an ENDCOR.

DEL,location,*location,I*

DEL deletes a record or records from the file being processed. If a second location is present, all records from the one specified in the first location through that of the second are deleted. If location represents an internal sequence number rather than the line number of the previous audit list, add I.

This control record is valid only after an EDIT and before an ENDCOR.

RPL,nnn,location,*location,I*

RPL deletes a record or records from the file being processed and replaces them with the following nnn data records. If a second location is present, all records from the one specified in the first through that of the second are deleted and replaced with the following nnn data records. If location represents an internal sequence number rather than the line number of the previous audit list, add I.

This control record is valid only after an EDIT and before an ENDCOR.

ENDCOR

ENDCOR indicates the end of modifications to the file in process. It copies the remainder of the file to be copied to the output device without modification.

SOURCE FILE

When a source file is to be copied or edited, it is input through the PI. The new or modified file is output by the PO.

A file in a source-language library consists of a header record, an EOF, the data records comprising the file, another EOF, and a catalog of all preceding files including the current one.

If a file follows another file, its header record immediately follows the catalog at the end of the preceding file. The last file on the source library is followed by two EOF records. Figure 6-3 shows the structure of a typical MOS source file.

HEADER RECORD

The header record is the first record of each file. It is generated with the file by FILE and is used as the identifier for the file. It is 14 words (28 characters) long in the following format:

```
1----------8    9----------20   21----------28
 FILENAME          AUTHOR             DATE
```

where the filename and author are taken from FILE parameters and date is the creation date as input on the /DATE card to the executive (Section 2).

DATA RECORD

A data record is a fixed-length record of 46 words in the following format:

```
1----------------80 81-------88     89------92
  CHARACTER DATA      BLANK           yddd
```

where yddd is the date (units position of year followed by the day) of the last change to the record (excluding resequencing).

CATALOG RECORD

The catalog record is a fixed-length record of eight words per entry, and up to 20 entries. Each catalog record contains an entry for each preceding file. Catalog records have the following format:

FILENAME

COUNT File 1

yddd

FILENAME

COUNT File 2

yddd

.
.
.

File n

where the filename is taken from FILE, count is a two-word binary integer representing the number of records in the file, and yddd is the date (units position of the year followed by the day) of the last change to the record (excluding resequencing).

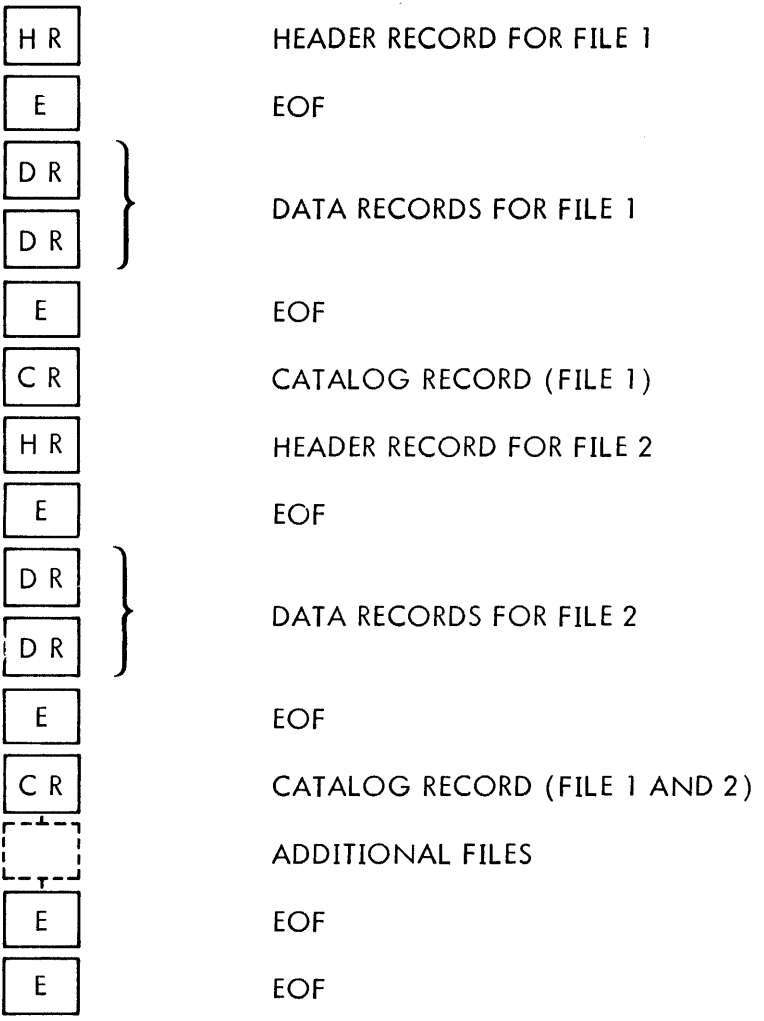


Figure 6-3. MOS Source File Structure

SECTION 7 - SYSTEM MAINTENANCE PROGRAM

The system maintenance program (SMP) is an MOS support program for use in updating an MOS installation system library (ISL) prior to the use of the ISL in system preparation. As input, the user supplies the SMP with control directives and an ISL. The program outputs a new library and optional directive listings of both old and new libraries.

Status and error messages are given in Section 13.

The SMP is loaded with /SMAIN (Section 2), which also specifies the logical input and output units. On loading, the program types the message:

```
BEGIN SYSTEM MAINTENANCE
```

on the Teletype (and LO, if different). All control directives are then input through SI before processing of the ISL is begun.

SMP directives have the following specifications:

a. General form:

   name,p(1),p(2),...,p(n)

   where name is the directive name and p(1),p(2),...p(n) is a parameter string with individual parameters separated by commas.

b. Directives have up to 72 characters and begin in the first character position of the record (card).

c. Embedded blanks are allowed within a field.

d. Fields smaller than the maximum are left-justified and blank-filled.

DIRECTIVES

ADD,name(1),name(2),...,name(n)

ADD adds a program or group of programs to the new ISL. The programs contained on the BI logical unit are added selectively to the new ISL after each program specified by ADD. Each name is the name of a program in the old ISL.

In processing ADD, the SMP copies from the old ISL to the new ISL until it has copied the program specified by a name in ADD. Then the SMP types the message:

```
ADD name
```

on the Teletype (and LO, if different). The program then waits for one of the following control directives from the Teletype:

Y Add the program contained on the BI, copy it into the new ISL, and return for another control directive.

N The additions are complete. Resume copying to the next name. N also types the message:

```
END ADDITIONS
```

DELETE,name(1),name(2),...,name(n)

DELETE deletes the specified program or programs from the old ISL when the new ISL is produced. Each name is the name of a program in the old ISL.

In processing DELETE, the SMP copies from the old ISL to the new ISL all programs except those named in DELETE.

REPLACE,name(1),name(2),...,name(n)

REPLACE replaces a program or programs from the old ISL. The program(s) specified in REPLACE are deleted and selectively replaced by the new program(s) contained on the BI. Each name is the name of a program in the old ISL.

In processing REPLACE, the SMP copies from the old ISL to the new ISL until it has copied the program preceding the program specified by a name in REPLACE. The SMP skips the specified program(s) on the old ISL and types the message:

```
REPLACE name
```

on the Teletype (and LO, if different). The program then waits for one of the following control directives from the Teletype:

Y Add the program contained on the BI, copy it into the new ISL, and return for another control directive.

N The replacements are complete. Resume copying to the next name. N also types the message:

```
END REPLACEMENTS
```

END,p

END initiates the ISL copying process. All necessary ADD, DELETE, and REPLACE directives are input before END. END rewinds the old and new ISL media, starts the copying process, and continues until it reaches the end of the old ISL.

END control directive parameter p is either:

L	List the old ISL on the LO
blank	Omit the listing (in this case, no comma follows END)

LIST

L or **LIST** lists on LO the ISL on the PI (Figure 7-1). A single letter (A or R), to the left of a program name under the ID header, identifies the program as an absolute or relocatable module.

DATE,xxxxxxxx

DATE sets up to eight alphanumeric characters (e.g., a date) in the page title of the list output. If DATE is not used, the date is used from the last/DATE input to the executive (section 2).

```
 PAGE     1  01/19/71

BEGIN SYSTEM MAINTENANCE
END,L



 PAGE     2  01/19/71

      ID        DATE     SIZE  ENTRY NAMES           EXTERNAL NAMES

 A PREP1     01/14/71 17435
 A PREP2     01/12/71 25377
ABS,RSCB1DK,05/12/70,-1,01000,010
 R RSCB1RMC 01/12/71 00721   CYLNO4 CYLNO1
                             CYLNOO   $XEQ
                              $UCB4  $UCB1
                              $UCBO   $TTL
                               $ROC   $REW
                               $PED   $PCW
                               $MPR   $LCB
                               $LBF   $LAX
                               $LAB   $JCW
                               $FRM   $ECW
                               $DAT   $BUF
ENDABS
ABS,RSCB1MT,01/11/70,-1,01000,010
 R RSCB1MTC            00662   CYLNO4 CYLNO1
                             CYLNOO   $XEQ
                              $UCB4  $UCB1
                              $UCBO   $TTL
                               $ROC   $REW
                               $PED   $PCW
                               $MPR   $LCB
                               $LBF   $LAX
                               $LAB   $JCW
                               $FRM   $ECW
                               $DAT   $BUF
ENDABS
ABS,RSCB2TY,01/11/70,-1,01000,010
 R RSCB2TY  03/19/70 00166     $XEQ              $ROC    $LAX    $JCW    $FRM
ENDABS
ABS,RSCB2LP,01/11/70,-1,01000,010
 R RSCB2LPB 10/29/70 00215     $XEQ              $ROC    $LBF    $LAX    $JCW
ENDABS
ABS,RSCB3,01/11/70,-1,01000,010
 R RSCB3               00435     IOCS    EXIT    $XEQ    $REW    $PUB    $LUB
                                EEXIT    $RST    $LAX    $JCW    $ECW    $BUF
                                 $PUN    $PGM
                                 $MOS    $LUN
                                 $LIT    $IOW
```

Figure 7-1. System Maintenance List Output

SECTION 8 - SYSTEM PREPARATION PROGRAM

The system preparation program (SPP) is a stand-alone support program of MOS. It creates a system file on a magnetic tape or rotating memory unit, tailored to the hardware and software requirements of the installation.

System preparation is controlled by SPP control directives. These directives include a description of the devices to be used for the system preparation, the devices to be included in the generated MOS system file, the system preparation functions to be executed, and the parameters for the system preparation functions.

Status and error messages are given in Section 13.

CONTROL DIRECTIVES

Control directives can be presented to the SPP from the SI in any order, except that the END directive must always be last. END signals the end of the SPP control directives and causes the program to begin generating the MOS system file on the PO.

SPP directives have the following specifications:

a. General form:

name,p(1),p(2),...,p(n)

where name is the directive name and p(1),p(2),...,p(n) is a parameter string separated by commas.

b. Directives have up to 72 characters and begin in the first character position of the record (card).

c. Blank records are ignored.

d. Embedded blanks are allowed within a field.

e. Numeric fields can be signed or unsigned octal or decimal integers. Octal numbers begin with a zero and decimal numbers do not.

f. Fields smaller than the maximum are left-justified and blank-filled.

DIRECTIVES:

ADD,name(1),name(2),...,name(n)

ADD adds a program or group of programs to the system file. The programs contained on the BI are added selectively to the system file after each program specified by ADD. Each name is the name of a program in the installation system library (ISL).

In processing ADD, the SPP copies from the ISL to the system file until it has copied the program specified by a name in ADD. Then, the SPP types the message:

```
ADD name
```

on the Teletype (and LO, if different). The program then waits for one of the following control directives from the Teletype:

Y Add the program contained on the BI, copy it to the system file, and return for another directive.

N The additions are complete. Resume copying to the next name. N also types the message:

```
END ADDITIONS
```

Note: The ADD control directive cannot be used to add a new program within an ABS-ENDABS program module.

DELETE,name(1),name(2),...,name(n)

DELETE deletes the specified program or programs from the ISL when the system file is produced. Each name is the name of a program in the ISL.

In processing DELETE, the SPP copies from the ISL to the system file all programs except those named in DELETE.

REPLACE,name(1),name(2),...,name(n)

REPLACE replaces the specified program or programs from the ISL with new programs contained on the BI. Each name is the name of a program in the ISL.

In processing REPLACE, the SPP copies from the ISL to the system file until it has copied the program preceding the program specified by name in REPLACE. The SPP skips the specified program on the ISL and types the message:

```
REPLACE name
```

on the Teletype (and LO, if different). The program then waits for one of the following control directives from the Teletype:

Y Add the program contained on the BI, copy it to the system file, and return for another directive.

N The replacements are complete. Resume copying to the next name. N also types the message:

```
END REPLACEMENTS
```

END,p(1),p(2)

END indicates to the SPP that there are no more system preparation control directives to process. Its parameters are:

L	List the ISL on the LO.
N	Do not list the ISL nor verify and list the new system file.
V	Verify the new system file.
blank	Do not list the ISL, but verify and list the new system file (in this case, no comma follows END).

DATE,xxxxxxxx

DATE sets up to eight alphanumeric characters (e.g., a date) in the page title of the list output. If DATE is not used, no date is printed.

COMP,p(1),p(2),p(3),p(4),p(5),p(6),p(7)

COMP defines the MOS computer system configuration as follows:

p(1)	Specifies the computer. If it is a 620/a or 622/a, p(1) is A. For all other Varian computers, P(1) is I or unspecified.
p(2)	Specifies the highest available memory address to be used for system preparation and the MOS system being prepared. On a system with 12K or larger memory, 025777 is assumed if the value is unspecified or less than 025777. On a system with 8K of memory, 017777 is assumed if the value is unspecified or less than 017777.
p(3)	Specifies the number (0, 1, 2, 3, or 4) of buffer interlace controllers (BICs) available in the system. If the value is unspecified or larger than four, zero is assumed. The SPP uses this parameter to supply the proper set of BIC library subroutines. If no BIC is available, dummy BIC library subroutines are entered in the system file. Otherwise, the program enters the actual BIC library subroutines.

p(4) Specifies the relocation bias set by the executive for the first program to be loaded by the loader if not specified in /LOAD (Section 2). A value of P(4)-1 is specified or less than the value of p(5), 04000 is assumed.

p(5) Specifies the base address set by the executive to be used by the loader for the direct literal pool if not specified in /LOAD (Section 2). If the value is unassumed if p(5) is either not specified, larger than 03777, or smaller than p(6).

p(6) Specifies the base address set by the executive to be used by the loader for the indirect address pool if not specified in /LOAD (Section 2). A value of 010 is assumed if p(6) is either not specified, smaller than 010, larger than 0777, or larger than p(5).

p(7) **Specifies the number of logical units in MOS. If it is unspecified or less than 14, a value of 14 is assumed. MOS handles up to 64 logical units.**

Commas must be present for unspecified parameters unless all parameters are unspecified. Thus, to define p(2) and p(7) only, input

```
COMP,,027777,,,,,25
```

and to set all parameters to their default values, input

```
COMP
```

IODEV,p(1),p(2),p(3)

IODEV adds or modifies entries in the MOS logical and physical unit tables as follows:

p(1) Specifies a four-character alphanumeric I/O device name. If /ASSIGN (Section 2) references this name, the SPP modifies the logical unit table to point to the physical unit.

p(2) Specifies the second and third characters of the three-character subroutine entry name of the I/O driver. The SPP stores these characters as the second two characters of the I/O driver in the physical unit table (the first character is always the dollar sign).

P(3) Specifies the relative position the I/O driver occupies in the physical unit table. The SPP enters the name of the I/O driver in the physical unit table in the order specified by p(3). If p(3) is omitted, the I/O driver is added to the physical unit table at the bottom.

Commas must be present for unspecified parameters unless all parameters are unspecified. Thus, to define p(1) and p(2) only, input

```
IODEV,MTU3,VW,
```

EQUIP,p(1),p(2),p(3),...,p(n)

EQUIP specifies the I/O devices to be included in the preparation of the system file. The SPP uses the parameters to construct the MOS logical and physical unit tables, and to select the proper drivers from the ISL when building the system file.

EQUIP parameters p(1) through p(n) are:

TYcu	Teletype keyboard/page printer
TRcu	Teletype paper tape reader
TPcu	Teletype paper tape punch
CRcu	Card reader
CPcu	Card punch
LPcu	Line printer
PTcu	High-speed paper tape reader and punch
PTcu(R)	High-speed paper tape reader only
PTcu(P)	High-speed paper tape punch only
MTcu(x)	Magnetic tape unit to be connected to BIC number x
MTcu	Magnetic tape unit
DRcu(x,y)	Drum memory unit to be connected to BIC number x; with y sectors allocated
DKcu(x,y)	Disc memory unit to be connected to BIC number x; with y sectors allocated

where

cu	specifies controller and unit number; if omitted, assume 00
P,R	are entered as key words (symbols)
X,Y	are numeric variables

Drum Partioning. Because of the low access time of the drum memory unit, MOS can partition the drum into from one to ten virtual units. Each virtual unit can function as a separate logical unit (except that the drum can perform only one operation at a time). Beginning and end-of-device sector addresses and a current address pointer for each unit are kept by MOS in the resident constant area. End-of-file marks are recorded on the first ten sectors of the drum.

The ten possible drum units are designated DR00 through DR09. EQUIP partitions the drum into the desired number of virtual units. A particular virtual unit is incorporated into the system if it is designated in an EQUIP parameter of the form **DRcu (x,y)**. Since x specifies the BIC, all drum units have the same x value. The number of sectors allocated to that virtual unit is given by y. If DR00 is designated as logical unit SF, y is the number of sectors assigned to DR00 in addition to those used by MOS. The value of x and y can be octal or decimal, but, if the former, it has a leading zero.

a. If DR00 is specified, it occupies sectors 012 to 012 + y if it is not the SF. If *DR00* is SF, it occupies sectors 012 to 012 + y + k, where k is the number of sectors in the system file.

b. Other specified virtual units begin at the end of the previous unit and occupy the next y sectors or the rest of the drum, whichever is smaller.

Since the size of the system file differs for each configuration, the number of sectors it occupies on DR00 is not known at the beginning of system preparation. The SPP, therefore, lists the sector allocation for each virtual unit on LO at the end of system preparation in the form:

```
DRUM ALLOCATION

DR00        000012      xxxxxx
DR01        yyyyyy      xxxxxx
.             .           .
.             .           .
.             .           .
DR09        yyyyyy      xxxxxx
```

where

xxxxxx	is the last sector address
yyyyyy	is the first sector address

All addresses are octal. Typical system files occupy about 03500 sectors.

If DR00 contains the MOS system file, the drum should be partitioned into at least two virtual units. The second unit is for intermediate storage of the source statements during assemblies and for other utility tasks. DR00 cannot contain both the MOS system file and assembly source statements.

Disc Partitioning. A disc MOS system can have one or two disc units. MOS can partition each disc unit into from one to ten virtual units. Each virtual unit can function as a separate logical unit (with the obvious exception that two virtual units on the same physical unit cannot function simultaneously). Beginning and end-of-device sector addresses and a current address pointer for each virtual unit are maintained by the system in the resident monitor.

The designations of the 30 possible virtual units, and the physical unit and controller corresponding to each, are shown below. EQUIP partitions the disc into the desired number of virtual units.

Controller	Disc Unit	Virtual Unit
0	0	DK00-DK09
0	1	DK10-DK19
1	0	DK40-DK49

A particular disc unit is incorporated into MOS if (and only if) one or more corresponding virtual unit designations appear in an EQUIP parameter of the form:

DKcu(x,y)

Since x specifies the BIC, DK00 through DK19 have the same n value. Similarly, DK40 through DK49 have the same n value.

The number of sectors to be allocated to that virtual unit is given by y. The value of y can be octal or decimal, but, if the former, it has a leading zero.

a. If DK00 is specified, it begins in sector 0 of the corresponding disc unit and occupies the next y sectors.

b. Each additional virtual unit specified for the same disc unit begins at the end of the previous virtual unit and occupies the number of sectors specified.

c. If the total number of sectors specified for all the virtual units is less than the total number on the disc, the last virtual unit is expanded to fill the rest of the disc. If the total number of sectors exceeds the total number on the disc, the first virtual unit to exceed the disc size is truncated to the capacity of the disc and all additional virtual units are zero sectors long.

d. If DK00 is SF, it begins in sector 0 and occupies the next y + k sectors, where k is the number of sectors required by the MOS system file.

Since the size of the system file differs for each configuration, the number of sectors it occupies on DK00 is unknown at the beginning of system preparation. The SPP, therefore, lists the sector allocation for each virtual unit on LO at the end of system preparation in the form:

```
DISC 0 ALLOCATION

DK00        000000      xxxxxx
DK01        yyyyyy      xxxxxx
.           .           .
.           .           .
.           .           .
DK09        yyyyyy      xxxxxx

DISC 1 ALLOCATION

DK10        000000      xxxxxx
DK11        yyyyyy      xxxxxx
.           .           .
.           .           .
.           .           .
DK19        yyyyyy      xxxxxx
```

where

xxxxxx is the last sector address
yyyyyy is the first sector address

All address are octal. Typical system files occupy about 03500 sectors.

If DK00 contains the MOS system file, the first disc unit is partitioned into at least two virtual units. The second unit is for intermediate storage of the source statements during assemblies, and for other utility tasks. DK00 cannot contain both the MOS system file and assembly source statements.

ASSIGN,l(1) = r(1),l(2) = r(2),...,l(n) = r(n)

Assign equates and assigns particular logical units to specific physical I/O devices. Execution of this directive decodes the parameter string and alters the logical unit table as specified by the parameter.

The parameters can be logical unit numbers, logical unit names, or physical unit names (Figure 3-1). In each parameter pair (i.e., each l(n) = r(n)), the left parameter, l(n), is a

logical unit number or name, and the right parameter, r(n), is a logical unit number or name or a physical device name.

In any case, the logical unit to the left of the equal sign is assigned to the unit/device to the right.

If r is a physical device, the l entry in the logical unit table is altered so that it points to the physical device driver specified by r. Thereafter, all I/O operations referencing l are directed to the physical device specified by r.

If r is a logical unit number or name, l is made equivalent to r and is assigned to the same physical device as r. However, if r is reassigned later to a new physical device, l no longer has an equivalent assignment.

The SPP makes default assignments for logical units not assigned to physical units. These assignments are a function of the physical devices included in the system. The SPP nominally assigns each logical unit the number of the highest peripheral device listed in table 8-1.

SF is always assigned to the same peripheral device that was used as PO in system preparation. SS is always assigned to the same logical unit as PO.

The first scratch unit is assigned to the first available device, the second scratch unit to the next available device, etc. If the end of the table is encountered, the table is rescanned from the top rather than assigning the logical unit to DUM. If no device is available, scratch units are assigned to DUM.

ABS,p(1),p(2),p(3),p(4),p(5)

ABS generates an absolute module on the system file. The input is an object module generated by assembly or compilation, preceded by the ABS. When the SPP encounters the ABS, the first object module that follows is converted into an absolute module and put on the MOS system file. Any following subprograms are only converted into the absolute module if referenced by the first (or a previous) program. No program or subprogram can contain an instruction reference to an externally defined literal.

The ABS parameters are:

p(1) An eight-character ASCII program identification name. The SPP stores this name in the identification block of the beginning absolute loader text record. If p(1) is omitted, the program generates a unique program identification name of the form XXnnnn for the program (where nnnn is a decimal number beginning at 0001 for the first program).

p(2) An eight-character creation date. The SPP stores p(2) in the date block of the beginning absolute loader text record.

Table 8-1. Valid SPP Logical Unit Assignments

SI	SO	PI	LO	BI	BO	GO	PO	S1,S2,S3,S4
TY00	TY00	CR00	LP00	CR00	CP00	MT02	MT01	MT01
TY10	TY10	CR10	LP10	CR10	CP10	DR02	DR01	DR01
CR00	DUM	PT00	TY00	PT00	PT00	MT03	MT02	MT02
CR10		PT10	TY10	PT10	PT10	DR03	DR02	DR02
DUM		TR00	DUM	TR00	TP00	MT10	MT03	MT03
		TR10		TR10	TP10	DR04	DR03	DR03
		DUM		DUM	DUM	MT11	MT10	MT10
						DR05	DR04	DR04
						MT12	MT11	MT11
						DR06	DR05	DR05
						MT13	MT12	MT12
						DR07	DR06	DR06
						MT20	MT13	MT13
						DR08	DR07	DR07
						MT21	MT20	MT20
						DR09	DR08	DR08
						MT22	MT21	MT21
						MT23	DR09	DR09
						MT30	MT22	MT22
						MT31	MT23	MT23
						MT32	MT30	MT30
						MT33	MT31	MT31
						DUM	MT32	MT32
							MT33	MT33
							MT00	
							DUM	

Figure 8-1. Valid SPP Logical Unit Assignments

p(3) A program relocation bias. The SPP uses this value to define the start of the first of the generated programs. If p(3) is omitted, the SPP uses the value of p(4) in COMP. If the p(3) is a -1, the SPP uses the current value of p(2) in COMP minus the size of this program as the relocation bias.

p(4) The base address of the direct literal pool. If the p(4) parameter is omitted, larger than 03777, larger than p(3), or smaller than p(5), the SPP uses the value of p(5) in COMP.

p(5) Specifies the base address of the indirect address pointers. If p(5) is omitted, smaller than 010, larger than 0777, or larger than p(4), SPP uses the value specified by COMP.

ENDABS

ENDABS terminates ABS processing. It follows the object modules input with the ABS.

INSTALLATION SYSTEM LIBRARY ORGANIZATION

The installation system library (ISL) is the primary input to the SPP. The order of the sections in the ISL and the programs within the sections must be maintained for proper MOS system file preparation and operation. The ISL sections are:

a. System preparation

b. System processor

c. System library

Comment records are alphameric records with a blank in the first character position. They are between, but not within, programs.

SYSTEM PREPARATION SECTION

This section contains:

a. Loader for the system preparation program

b. System preparation program part 1

c. System preparation program part 2

SYSTEM PROCESSOR SECTION

This section contains:

a. Resident system configuration block part 1 (resident monitor)

b. Resident system configuration block part 2 (dump)

c. Resident system configuration block part 3 (I/O control)

d. Resident system configuration block part 4 (executive)

e. Loader

f. Loader system file I/O drivers

g. Loader map subroutine

h. Loader list output I/O drivers

i. Loader binary input I/O drivers

j. DAS MR assembler

k. FORTRAN IV compiler

l. Debugging program

m. Concordance program

n. System maintenance program

o. File editing program

p. Input/output drivers

SYSTEM LIBRARY SECTION

This section comprises the FORTRAN IV run-time library and programs that perform utility functions. Utility or often-used object programs can be added and executed with /ULOAD (section 2). The final programs in this section are the I/O drivers. All user programs precede the I/O drivers.

OPERATING PROCEDURES

To prepare a MOS system:

a. Load the system preparation program (SPP) by entering the proper bootstrap program into the computer memory.

b. Assign the peripheral devices for use by the SPP.

c. Supply the control directives to define a MOS system file.

LOADING

Depending on the peripheral device used to read the SPP, one of the following initialization procedures applies:

a. Card reader
   (1) Turn on the card reader.
   (2) Place two blank cards after the last control-directive card of the ISL deck.
   (3) Place the ISL deck in the card hopper.
   (4) Press clear and start.

b. 33/35 ASR Teletype
   (1) Turn on the Teletype
   (2) Place Teletype in off-line mode and simultaneously press the CONTROL and D, then CONTROL and T, and finally the CONTROL and Q keys.
   (3) Position the system preparation loader program paper tape in the reader with the first binary frame at the reading station. Close the reading gate.
   (4) Set the reader control level to STOP, and the Teletype on-line.

c. High-speed paper tape reader
   (1) Turn on the paper tape reader.
   (2) Position the system preparation loader program paper tape in the reader with the first binary frame at the reading station. Close the reading gate.
   (3) Set the LOAD/RUN switch to RUN.

d. Magnetic tape unit
   (1) Turn on the magnetic tape unit.
   (2) Mount the ISL magnetic tape.
   (3) Position the magnetic tape to the load point.
   (4) Ready the magnetic tape unit so it can be used by the computer.

Enter the appropriate bootstrap loading routine (tables 8-2 and 8-3). Depending on the Varian computer used, one of the following procedures apply:

a. V73 Computer
   1. Load the starting memory address of the bootstrap loader (007756) into the P register.
   2. Press MEM switch momentarily.
   3. Clear the console display (Press DISPL CLR).
   4. Select the first bootstrap loader instruction and load it into the control-panel display register.
   5. Press ENTER to load the display-register contents into the address specified by the P register, which is incremented by one after the instruction is loaded.
   6. Clear the display register (Press DISPL CLR).
   7. Repeat steps 3, 4, 5, and 6 for each bootstrap loader instruction.

b. 620/f and 620/f-100 Computers
   1. In step mode, load a store A register relative to P instruction (054000) into the instruction register.
   2. Set the REPEAT switch.
   3. Load the starting memory address of the bootstrap loader into the P register.
   4. Select the first bootstrap loader instruction and load it into the A register.
   5. Press STEP or START to load the A register contents into the address specified by the P register, which is incremented by one after the instruction is loaded.
   6. Clear the A register.
   7. Repeat steps 4, 5, and 6 for each bootstrap loader instruction.

c. 620/L and 620/L-100 Computers
   1. In step mode load a store A register relative to P instruction (054000) into the instruction register.
   2. Press the REPEAT switch.
   3. Load the starting memory address of the bootstrap loader into the P register.
   4. Select the first bootstrap loader instruction and load it into the A register.
   5. Press STEP to load the A register contents into the address specified by the P register, which is incremented by one after the instruction is loaded.
   6. Clear the A register by pressing BIT RESET.
   7. Repeat steps 4, 5, and 6 for each bootstrap loader instruction.

Initiate the bootstrap from the peripheral device as follows:

a. To initiate loading from the card reader, high-speed paper tape reader, or magnetic device, reset the A, B, X, P, and instruction registers; then, press SYSTEM RESET and RUN (for V73 and 620/f press RESET, position STEP/RUN to RUN, and press START).

   The system preparation loader and ISL are separate paper tapes. The ISL must be mounted when the computer goes to STEP after reading the loader.

b. To initiate loading from the Teletype, follow step a; then, set the reader control level to RUN. (Start position on ASR 33.)

NOTE

If an error occurs while loading the system preparation modules, the computer goes to the STEP mode with the instruction register = 0777 and A = B = X = −1. Recovery is made by repositioning the last record read at the read station and press RUN (START for V73 and 620/f). For magnetic tape, the repositioning is automatic.

ASSIGNMENT

At the end of a successful loading, the Teletype makes five requests for peripheral device assignments to be used by the SPP. The form of these requests is:

Table 8-2. 620 Bootstrap Loading Routines (16-Bit Computer)

Address	Magnetic Tape Controller 0 Unit 0	Magnetic Tape Controller 0 Unit 1	Magnetic Tape Controller 1 Unit 0	33/35 ASR Teletype	High-Speed Paper Tape Reader	Card Reader
00000	104110	104210	104111	102601	100537	100230
00001	101210	101210	101211	030011	030011	101130
00002	000005	000005	000005	005101	005101	000007
00003	001000	001000	001000	101201	101537	101630
00004	000001	000001	000001	000007	000007	0xx400
00005	030020	030020	030020	001000	001000	001000
00006	100010	100010	100011	000003	000003	000001
00007	102510	102510	102511	102601	102637	102230
00010	055000	055000	055000	001020	001020	030024
00011	005144	005144	005144	0xx400	0xx400	004244
00012	101110	101110	101111	004050	004050	004344
00013	000007	000007	000007	004002	004002	004444
00014	101210	101210	101211	004446	004446	060030
00015	0xx401	0xx401	0xx401	001020	001020	020027
00016	001000	001000	001000	000003	000003	004142
00017	000012	000012	000012	055000	055000	056000
00020	0xx400	0xx400	0xx400	005144	005144	005344
00021				001000	001000	040027
00022				000002	000002	020030
00023						001040
00024						000003
00025						001000
00026						000011
00027						yy5777

xx = 17 for 8K systems; xx = 25 for 12K or larger systems
yy = 07 for 8K systems; yy = 12 for 12K or larger systems

Table 8-3. Bootstrap Loading Routine (18-Bit Computer)

Address	Magnetic Tape Controller 0 Unit 0	Magnetic Tape Controller 0 Unit 1	Magnetic Tape Controller 1 Unit 0	33/35 ASR Teletype	High-Speed Paper Tape Reader	Card Reader
00000	104110	104210	104111	102601	100537	100230
00001	101210	101210	101211	030011	030011	001000
00002	000005	000005	000005	005101	005101	000006
00003	001000	001000	001000	101201	101537	0xx377
00004	000001	000001	000001	000007	000007	040003
00005	030020	030020	030020	001000	001000	067003
00006	100010	100010	100011	000003	000003	040003
00007	102510	102510	102511	102601	102637	006010
00010	055000	055000	055000	001020	001020	001036
00011	005144	005144	005144	0xx400	0xx400	050022
00012	101110	101110	101111	004052	004052	005007
00013	000007	000007	000007	004002	004002	101630
00014	101210	101210	101211	004446	004446	0xx400
00015	0xx401	0xx401	0xx401	001020	001020	101130
00016	001000	001000	001000	000003	000003	000021
00017	000012	000012	000012	055000	055000	001000
00020	0xx400	0xx400	0xx400	005144	005144	000013
00021				001000	001000	102230
00022				000002	000002	001036
00023						000004
00024						004454
00025						057003
00026						040022
00027						001000
00030						000013

xx = 17 for 8K systems; xx = 25 for 12K or larger systems

```
ENTER DEVICE NAME FOR xx
```

where xx denotes PI, PO, LO, BI, and SI, respectively, in the five requests. In response to each request, type the name of a peripheral device followed by a carriage return. The specified peripheral device is assigned the corresponding logical function during the system preparation process.

Table 8-4 gives the function of each logical unit during system preparation and lists acceptable peripheral device assignments for each logical unit name. To assign the peripheral device a default assignment, type a carriage return.

After completing peripheral device assignments for system preparation, the message:

```
BEGIN SYSTEM PREPARATION
```

is output. The SPP can then accept system preparation control directives defining the MOS system file to be generated.

DISC FORMATTING

When preparing an MOS system for a 620-40 disc, a disc-formatting routine is loaded by the SPP loader. This routine is loaded prior to the SPP.

At the end of a successful loading, the Teletype makes four requests for disc formatting information. These requests are for the BIC hardware address, controller hardware address, disc unit number, and the disc size (in cylinders). Each input can be either a decimal or octal number, but, if the latter, it has a leading zero. Each request is terminated by a carriage return.

After the fourth request, the routine formats the disc. When formatting is complete, formatting information for another disc is requested. When all discs have been formatted, answer the request for the BIC hardware address by pressing the CONTROL and BELL keys on the Teletype. The loader then loads the next segment of the SPP and proceeds as described in section 4.

SYSTEM VERIFICATION AND COMPLETION

Upon completion of the system preparation, the system file is read and verified to ensure that no errors occurred. Verification consists of reading the new system file and checking such items as checksums, sequence numbers, and record formats. A listing is also generated on the LO (figure 8-1). If verification is not desired, supply the parameter N on the END control directive. After the system preparation is complete, the program outputs the message:

```
MOS SYSTEM READY
```

It then loads the resident monitor into memory by executing the appropriate bootstrap of the MOS system file and types ** on the Teletype.

Table 8-4. Logical Unit Functions

Unit	Function	Assignments	Default
PI	Contains the ISL	TR00***** PT00 CR00* MT01**** MT10**** MT00****	TR00
PO	Contains the MOS system	MT00**** DR00** DK00***	MT00
LO	Lists all SPP control directives and the ISL or MOS system file, if requested	TY00 LP00	TY00
BI	Inputs any additional programs not contained on the ISL	TR00 PT00 CR00*	TR00
SI	Inputs all SPP control directives	TY00 TR00 PT00 CR00*	TY00

*Only on systems with 12K or larger memory.

**DR00 is acceptable only on the drum and magnetic tape MOS.

***DK00 is acceptable only on the disc MOS.

****MT00 is acceptable on systems with 12K or larger memory or any drum and magnetic tape MOS.

*****When TR00 is specified, TP00 must also be specified within the EQUIP directive parameter.

```
 PAGE     1

BEGIN SYSTEM PREPARATION
DATE,01/20/71
COMP,I,025777,1,0500,0477,010,14
EQUIP,MT00,MT10,TY,TR,TP,PT,CR,LP
EQUIP,DK00(022,0),DK01(022,02000),DK02(022,01000),DK03(022,01000)
END,L
```

Figure 8-1. System Preparation List Output (1 of 3)

```
 PAGE    2  01/20/71

       ID        DATE     SIZE  ENTRY NAMES           EXTERNAL NAMES

 A BOOTSTRP 01/20/71 00442
ABS,RSCR1DK,05/12/70,-1,01000,010
   $PUB   25006
   $LUB   25037
CYLN04    25766
CYLN01    25766
CYLN00    25740
   $XEQ   25066
  $UCB4   25741
  $UCB1   25741
  $UCB0   25713
   $TTL   25072
   $ROC   25776
   $REW   25505
   $PED   25070
   $PCW   25056
   $MPR   25146
   $LCB   25057
   $LBF   00400
   $LAX   25255
   $LAB   25270
   $JCW   25057
   $FRM   25102
   $ECW   25071
   $DAT   25076
   $BUF   25617
[$IAP]    00010
[$LIT]    01000
[$PED]    25055
ABS,RSCR1MT,01/11/70,-1,01000,010
ENDABS
ABS,RSCR2TY,01/11/70,-1,01000,010
ENDABS
ABS,RSCR2LP,01/11/70,-1,01000,010
   $PUB   25006
   $LUB   25037
CYLN04    25766
CYLN01    25766
CYLN00    25740
   $XEQ   25066
```

Note: If a slash appears to the left of a number field, the designated program is required by the system preparation program but omitted from the ISL.

Figure 8-1. System Preparation List Output (2 of 3)

```
PAGE    9  01/20/71

      ID        DATE     SIZE  ENTRY NAMES          EXTERNAL NAMES

                              $LBF    $LAX
                              $LAB    $JCW
                              $IOW    $IAP
                              $FRM    $ECW
                              $DAT    $COR
                              $COM    $BUF
                              $ALG
  R MAP$                   00123  MAP$          LCB$    IOCS    $BUF
  R $00                    00022    $00         MWR$    MT$0    MSR$    MSF$
                                                MRW$    MRD$    MEF$    MCK$
 *R $01                    00022    $01         MWR$    MT$0    MSR$    MSF$
                                                MRW$    MRD$    MEF$    MCK$
 *R $02                    00022    $02         MWR$    MT$0    MSR$    MSF$
                                                MRW$    MRD$    MEF$    MCK$
 *R $03                    00022    $03         MWR$    MT$0    MSR$    MSF$
                                                MRW$    MRD$    MEF$    MCK$
  R $04                    00022    $04         MWR$    MT$1    MSR$    MSF$
                                                MRW$    MRD$    MEF$    MCK$
 *R $05                    00022    $05         MWR$    MT$1    MSR$    MSF$
                                                MRW$    MRD$    MEF$    MCK$
 *R $06                    00022    $06         MWR$    MT$1    MSR$    MSF$
                                                MRW$    MRD$    MEF$    MCK$
 *R $07                    00022    $07         MWR$    MT$1    MSR$    MSF$
                                                MRW$    MRD$    MEF$    MCK$
 *R $08                    00022    $08         MWR$    MT$2    MSR$    MSF$
                                                MRW$    MRD$    MEF$    MCK$
 *R $09                    00022    $09         MWR$    MT$2    MSR$    MSF$
                                                MRW$    MRD$    MEF$    MCK$
 *R $0A                    00022    $0A         MWR$    MT$2    MSR$    MSF$
                                                MRW$    MRD$    MEF$    MCK$
 *R $0B                    00022    $0B         MWR$    MT$2    MSR$    MSF$
                                                MRW$    MRD$    MEF$    MCK$
 *R $0C                    00022    $0C         MWR$    MT$3    MSR$    MSF$
                                                MRW$    MRD$    MEF$    MCK$
 *R $0D                    00022    $0D         MWR$    MT$3    MSR$    MSF$
                                                MRW$    MRD$    MEF$    MCK$
 *R $0E                    00022    $0E         MWR$    MT$3    MSR$    MSF$
                                                MRW$    MRD$    MEF$    MCK$
 *R $0F                    00022    $0F         MWR$    MT$3    MSR$    MSF$
                                                MRW$    MRD$    MEF$    MCK$
```

Note: An asterisk preceeding a line indicates that the designated program is contained in the ISL but omitted from the created System File.

Figure 8-1. System Preparation List Output (3 of 3)

If the system file is on a rotating memory unit, the virtual unit allocation table is listed on SO followed by the message:

```
DISC MOS SYSTEM READY
```

EXAMPLES

Problem 1

Prepare an MOS system file for a 620/622 computer system having one magnetic tape unit, one Teletype unit, and 8K of core memory. Make the following logical unit assignments: SF = MT00, PI = TR00, SS = MT00, BI = TR00, SO = TY00 LO = TY00, and BO = TP00. Set the default values of $PGM to 03000, $LIT to 02777, and $IAP to 010. During system preparation, do not list the ISL, but verify and list the MOS system file.

Procedure:

1. Key in the Teletype bootstrap loader and enter the SPP through the Teletype paper tape reader.

2. Assign logical units for use by the SPP as follows:

   PI = TR00
   PO = MT00
   LO = TY00
   BI = TR00
   SI = TY00

3. Mount the ISL on the Teletype reader (PI) and mount a scratch magnetic tape with a write-ring on the magnetic tape transport (PO).

4. Respond to the **BEGIN SYSTEM PREPARATION** message with the following control directives on the Teletype keyboard (SI):

```
COMP,I,017777,0,03000,02777,010,14
DATE,11/07/69          (optional)
EQUIP,TY,MT,TR,TP
END
```

Problem 2

Prepare an MOS system file for a 620/622 computer system having four standard magnetic tape units connected to one controller, one special magnetic tape unit connected to a different controller for which the user supplies the I/O driver, one Teletype unit, one line printer, one card reader, one high-speed paper tape reader/punch unit, and 16K of core memory. The special magnetic tape unit is designated MM10 and its I/O driver has the entry name $0U. Make the following logical unit assignments: SF = MT00, PO = MT01, SS = MT01, GO = MT02, SI = TY00, PI = CR00, LO = LP00, BI = PT00, BO = PT00, S1 = MT03, and LUN 15 = MM10 (special magnetic tape unit). Set the default values of $PGM to 04000, $LIT to 03777, and $IAP to 030. During the system preparation process, replace the FORTRAN compiler with a new version and verify and list both the ISL and the MOS system file.

Procedure:

1. Key in the magnetic tape unit 2 bootstrap and enter the SPP through MT01.

2. Assign logical units for use by the SPP as follows:

   PI = MT01
   PO = MT00
   LO = LP00
   BI = PT00
   SI = TY00

3. Mount the ISL on MT01 (PI) and mount a magnetic tape on MT00 (PO)

4. Respond to the **BEGIN SYSTEM PREPARATION** message by typing the following control directives on the Teletype (SI):

```
COMP,I,037777,0,04000,03777,030,15
DATE,11/07/69          (optional)
IODEV,MM10,0U,5
EQUIP,TY,CR,LP,PT,TR,TP
EQUIP,MT00,MT01,MT02,MT03,MM10
ADD,$03
REPLACE FORTRAN
ASSIGN,BI=PT00,BO=BI
assign, S1 = MT03, 15 = MM10


END,L
```

5. When the SPP types the message:

```
ADD $03
```

place the special magnetic tape I/O driver object program ($0U) in the high-speed paper tape reader. Type Y to copy $0U on the MOS system file. After the program has been copied, type N to continue system preparation.

6. When the SPP types the message:

   ```
   REPLACE FORTRAN
   ```

   place the new FORTRAN compiler program on the high-speed paper tape reader. Type Y to copy the FORTRAN compiler to the MOS system file. After the FORTRAN compiler has been copied, type N to continue system preparation.

Problem 3:

Prepare an MOS system file for a 620/622 computer system having one 256-track drum memory unit (connected to BIC 020), two standard magnetic tape units with separate controllers, one Teletype unit, one line printer, one high-speed paper tape reader/punch unit, and a 32K core memory. Prepare the system so that only the first 31K of core is used. Make the following logical unit assignments: SF = DR00, PI = PT00, SS = DR01, PO = DR01, BI = PT00, SI = TY00, SO = TY00, LO = LP00, and BO = PT00. Set the default values of $PGM to 0500, $LIT to 0377, and $IAP to 010. Partition the drum memory into six virtual units. Make the first unit 050 sectors longer than the system file, and the next five units 05000, 03000, 01500, 01500, and 01500 sectors long, respectively. Do not list the ISL and do not verify or list the MOS system file.

Procedure:

1. Mount the ISL on MT00 (PI).

2. Key in the magnetic tape unit 1 bootstrap and enter the SPP through MT00.

3. Assign logical units for use by the SPP as follows:
   PI = MT00
   PO = DR00
   LO = LP00
   BI = PT00
   SI = TY00
4. Respond to the **BEGIN SYSTEM PREPARATION** message by typing the following control directives on the Teletype (SI):

```
COMP,I,075000,1,0500,0377,010,14
DATE,05/26/70          (optional)
EQUIP,MT00,MT10,TY,TR,TP,PT,LP
EQUIP,DR00(020,050),DR01(020,05000)
EQUIP,DR02(020,03000),DR03(020,01500)
EQUIP,DR04(020,01500),DR05(020,01500)
END,N
```

5. A drum allocation listing of the following form is printed on LO after system preparation is complete:

```
DRUM ALLOCATION

DR00        000012        003477
DR01        003500        010477
DR02        010500        013477
DR03        013500        015177
DR04        015200        016677
DR05        016700        017777
```

Problem 4:

Prepare an MOS system file for a 620/622 computer system having one disc unit connected to BIC 020, one disc unit connected to BIC 022, two 9-track magnetic tape units connected to BIC 024, one Teletype unit, one card reader, one line printer, and 32K core memory. Prepare the system so that only the first 28K of core is used. Make the following logical unit assignments: SF = DK00, PI = CR00, PO = DK01, SS = DK01, BI = MT01, SI = TY00, SO = TY00, LO = LP00, BO = MT00, S1 = DK02, S2 = DK03, S3 = DK40, and S4 = DK41. Partition the first disc unit into four virtual units, with the first virtual unit equal in length to the system file (SF) plus 0100 sectors and the other three virtual units 01500, 01000, and 01000 sectors long, respectively. Partition the second disc unit into five virtual units, each 01200 sectors long. Set the default values of $PGM to 02000, $LIT to 01777, and $IAP to 0500. During system preparation, do not list the ISL, but verify and list the MOS system file.

Procedure:

1. Mount the ISL on MT00 (PI).

2. Key in the magnetic tape unit 0 bootstrap and enter the SPP through MT00.

3. Assign logical units for use by the SPP as follows:

```
PI = MT00
PO = DK00
LO = LP00
BI = MT01
SI = TY00
```

4. Respond to the **BEGIN SYSTEM PREPARATION** message by typing the following control directives on the Teletype (SI):

```
COMP,I,067777,3,02000,01777,0500,14
DATE,06/11/70              (optional)
EQUIP,MT00(024),MT01(024),TY,CR,LP
```

```
EQUIP,DK00(020,0100),DK01(020,01500),DK02(020,01000)
   DK03 (020,01000)
EQUIP,DK40(022,01200),DK41(022,01200),DK42(022,01200)
EQUIP,DK43(022,01200),DK44(022,01200)
ASSIGN,PO=DK01,SS=DK01,BI=MT01,BO=MT00
ASSIGN,S1=DK02,S2=DK03,S3=DK40,S4=DK41
END
```

A disc allocation listing of the following form is printed on LO after system preparation is complete:

```
DISC 0 ALLOCATION

DK01        000000      003577
DK01        003600      005277
DK02        005300      006257

DISC 4 ALLOCATION

DK40        000000      001177
DK41        001200      002377
DK42        002400      003577
DK43        003600      004777
DK44        005000      006257
```

PROBLEM 5:

Prepare a MOS system file for a 620/622 computer system having one standard magnetic tape unit connected to BIC device address 022, one Teletype unit, one card reader, one high-speed paper tape reader/punch unit, one movable-head disc unit with 9,744 sectors connected to BIC device address 020, one STATOS 21 (620-74) printer/plotter to be used as the system line printer, and 32K core memory. The STATOS 21 printer/plotter programs are not resident on the Installation System Library (ISL) and it is desired to replace the standard line printer routines " RSCB2LPB" (core dump), " $0Q" (DST), and " LPDP24" (line printer driver) with the STATOS 21 routines " RSCB2LPB" , " $0Q" , and " LPST21" . Set the default values of $PGM to 0500, $LIT to 0377, $IAP to 010, and number of logical units to twenty. Partition the disc unit into ten virtual units, with the first virtual unit equal in length to the system file (SF), the second virtual unit through the tenth virtual unit into 3000, 800, 500, 900, 700, 100, 300, 200, and 100 sectors long, respectively. Do not list the ISL, but verify the MOS system file.

Procedure:

NOTE:

LO cannot be assigned to LP00 unless SPP contains the appropriate line printer driver.

1. Mount the ISL on MT00 (PI).

2. Key in the magnetic tape unit 0 bootstrap and enter the SPP through MT00.

3. Prepare the system preparation control directives on punched cards and place into the input hopper. Make the card reader ready.

4. Assign logical units for use by the SPP as follows:

   PI = MT00
   PO = DK00
   LO = TY00
   BI = CR00
   SI = CR00

5. Be prepared to place the appropriate binary-object program replacements into the card reader when the SPP requests the replacements for " RSCB2LPB", " $0Q", and " LPDP24"

6. Respond to the **Begin System Maintenance** message by making the card reader ready. The following directives, previously prepared, will be read and printed on the Teletype (LO)

```
DATE,07-27-71 (optional)
COMP,I,075777,2,0377,010,20
EQUIP,TY,PT,LP,CR,MOT00(022),DK00(020,0),DK01
(020,3000)
EQUIP,DK02(020,800),DK03(020,500),DK04(020,900),
DK05(020,700)
EQUIP,DK06(020,100),DK07(020,300),KD08(020,200),
DK09(020,100)
REPLACE,$0Q,LPDP24,RSCB2LPB
END,V
```

7. When the SPP types the message:

   REPLACE RSCB2LPB

   place the STATOS 21 core dump routine object program (RSCB2LPB) in the card reader and make ready. Type Y to copy RSCB2LPB onto the MOS system file. After the program has been copied, type N to continue system preparation.

8. When the SPP types the message:

   REPLACE $0Q

   place the STATOS 21 I/O driver DST object program ($0Q) in the card reader and make ready. Type Y to copy $0Q onto the MOS system file. After the program has been copied, type N to continue system preparation.

9. When the SPP types the message:

REPLACE LPDP24

place the STATOS 21 I/O driver object program (LPST21) in the card reader and make ready. Type Y to copy LPST21 onto the MOS system file. After the program has been copied, type N to continue system preparation.

10. Repeat steps 8 and 9 until all required replacements are completed.

11. A disc allocation listing of the following form is printed on LO (TY00) after system preparation is complete:

```
DISC 0 ALLOCATION

DK00        000000      004544
DK01        004545      012434
DK02        012435      014074
DK03        014075      015060
DK04        015061      016664
DK05        016665      020160
DK06        020161      020324
DK07        020325      021000
DK08        021001      021310
DK09        021311      023017
```

12. After the system has been verified by SPP the following is printed on the LO:

MOS SYSTEM READY

13. The system file is then bootstrapped into the computer via the memory resident MOS and the following is printed on the Teletype (SO):

* *

SECTION 9 - LANGUAGE PROCESSORS

The basic MOS supports the DAS MR assembler as its language processor. By increasing memory, the FORTRAN IV (ANSI standard) compiler can also be used. The modular design of the MOS allows the inclusion of additional Varian or user language processors through the system preparation procedure.

Both the assembler and compiler exist in stand-alone and MOS configurations. This chapter describes the features of the MOS versions of these language processors.

Status and error messages are given in Section 13.

DAS MR ASSEMBLER

DAS MR is a two-pass macro assembler that uses the secondary storage device of MOS for the Pass 1 output. It reads a source module from the PI and outputs it on the PO. The Pass 2 source input is input from the SS.

When an END statement is encountered, the SS is repositioned and reread. During Pass 2, the output can be directed to the BO for the object module and the LO for the assembly listing. The SS or PO file, which contains a copy of the source module, can be used as input to a subsequent assembly.

A DAS MR symbol consists of one to six characters, the first of which must be alphabetic, with the rest alphabetic or numeric. Additional alphanumeric characters can be appended to the first six characters of the symbol to form an extended symbol up to the limit imposed by a single line of code. However, only the first six characters are recognized by the assembler.

Assembler language programs (and subroutines) are referred to within MOS by name during system maintenance and system preparation. To name a DAS MR module in MOS, specify from one to eight characters in the title parameter of the /JOB preceding the /ASSEMBLE control directive.

The DAS MR assembler provides for relocatable and absolute object modules. Absolute modules must be explicitly specified with an ORG directive. Otherwise, the modules will be relocatable.

The directives recognized by the DAS MR assembler are:

BES	DATA	END	GOTO	MZE	PZE
BSS	DETL	ENTR	IFF	NAME	RETU*
CALL	DUP	EQU	IFT	NULL	SET
COMN	EJEC	EXT	LOC	OPSY	SPAC
CONT	EMAC	FORM	MAC	ORG	SMRY

FORTRAN IV COMPILER

The FORTRAN IV compiler is a one-pass compiler. It inputs a source module from the PI and produces an object module on the BO and an object listing on the LO. No secondary storage is required for a compilation.

When a fatal error is detected (T type, section 13), the compiler automatically terminates the BO. LO output continues. The compiler reads from the PI file until an END statement is encountered or a control directive is read. Compilation also terminates on detection of an I/O error or an end-of-device, beginning-of-device, or end-of-file indication from I/O control.

FORTRAN IV programs (subroutines, functions, block data, etc.) are referred to within MOS by name during system maintenance and system preparation. To name a FORTRAN IV module in MOS, specify from one to eight characters in the title parameter of the /JOB preceding the /FORTRAN control directive.

The FORTRAN IV compiler output comprises relocatable object modules under all circumstances (e.g., main programs, subprograms, functions, etc.).

The FORTRAN IV compiler has conditional compilation facilities implemented by an X in column 1 of a source statement. When the X appears in the /FORTRAN directive, all source statements with an X in column 1 are compiled with all other statements (viz., the X is treated as a blank). When the X is not present, all conditional statements are ignored by the compiler. X lines are given numbers on the list output in either case, but the source statement is printed only when the X is present.

When performing I/O in FORTRAN, the READ and WRITE statements are used. In these statements, I/O devices are referenced by logical unit numbers. The MOS FORTRAN IV supports logical units 1 through 255 (table 3-1). These logical unit numbers are assigned to physical devices (table 3-2).

RPG IV

The MOS RPG IV system is a software package for general data processing applications. It combines versatile file and record defining capabilities with powerful processing statements to solve a wide range of applications. It is particularly useful in the processing of data for reports. The MOS RPG IV system consists of an RPG IV compiler and a runtime/loader program.

The RPG, Report Program Generator, its compilation and execution under MOS are described in the Varian RPG IV User's Manual (98 A 9947 03R, where R is the revision number).

SECTION 10 - SUPPORT LIBRARY

The MOS system has a comprehensive subroutine library directly available to the user. The library contains mathematical subroutines to support the execution of a FORTRAN IV program, plus many commonly used utility subroutines. To use the library, merely code the proper call in the program, or, for the standard FORTRAN IV functions, implicitly reference the subroutine (e.g., A = SQRT(B) generates a CALL SQRT(B)). All calls generate a reference to the required routine, and the loader brings the subroutine into memory and links it to the calling program.

CALLING SEQUENCE

The subroutines in the support library can be called through a DAS MR or FORTRAN IV program as follows:

DAS MR

General form:

```
label CALL S,p(1),p(2),...,p(n)
```

Expansion:

```
label     JMPM      S
          DATA      p(1)
          DATA      p(2)
            .
            .
            .
          DATA      p(n)
```

FORTRAN IV

General form:

```
statement number  CALL S(p(1),p(2),...,p(n))
```

Generated code:

```
JMPM        S
DATA        p(1)
DATA        p(2)
  .
  .
  .
DATA        p(n)
```

SIXTEEN-BIT NUMBERS

Single-precision integers use one 16-bit word. A negative number is in two's complement form. An integer in the range -32,768 to +32,767 can be stored as a single-precision integer.

Single-precision floating-point numbers use two consecutive 16-bit words. The exponent (in excess 0200 form) is in bits 14 to 7 of the first word. The mantissa is in bits 6 to 0 of the first word and bits 14 to 0 of the second word. The sign bit of the second word is always zero. A negative number is represented by the one's complement of the first word. Any real number in the range $10^{\pm 38}$ can be stored as a single-precision floating-point number having a precision of six digits.

Single-Precision Floating-Point Number (16-Bit)

```
Bit   15    14 13 12    11 10 9    8 7 6    5 4 3    2 1 0
n)    s     ------Exponent---------- ---High Mantissa-
n+1)  0     ---------------Low Mantissa---------------
```

Double-precision floating-point numbers use four consecutive 16-bit words. The exponent (in excess 0200 form) is in bits 7 to 0 of the first word. The mantissa is in the second, third, and fourth words. Bit 17 of the third and fourth words and bits 17 to 8 of the first word are zero. A negative number is represented by the one's complement of the second word. Any real number in the range $10^{\pm}$ can be stored as a double-precision floating-point number having a precision of 13 decimal digits.

Double-Precision Floating-Point Numbers (16-bit)

```
Bit   15    14 13 12    11 10 9    8 7 6    5 4 3    2 1 0
n)    0     0  0  0     0  0  0    0 ------Exponent-----
N+1)  S     --------------High Mantissa---------------
n+2)  0     --------------Mid Mantissa----------------
n+3)  0     --------------Low Mantissa----------------
```

EIGHTEEN-BIT NUMBERS

Single-precision integers use one 18-bit word. A negative number is in two's complement form. Any integer in the range -131,072 to +131,071 can be stored as a single-precision integer.

Single-precision floating-point numbers use two consecutive 18-bit words. The exponent (in excess 0200 form) is in bits 16 to 9 of the first word. The mantissa is in bits 8 to 0 of the first word, and bits 16 to 0 of the second word. The sign bit of the second word is always zero. A negative number is represented by the one's complement of the first word. Any real number in the range -76,000,000,000 to +76,000,000,000 can be stored as a single-precision floating-point number having a precision of seven digits.

Single-Precision Floating-Point Number (18-Bit)

```
Bit   17 16 15   14 13 12   11 10 9   8 7 6   5 4 3   2 1 0
n)    s  ---------Exponent-------  -----High Mantissa-
n+1)  0  ------------------Low Mantissa----------------
```

Double-precision floating-point numbers use four consecutive 18-bit words. The exponent (in excess 0200 form) is in bits 7 to 0 of the first word. The mantissa is in the second, third, and fourth words. Bit 17 of the third and fourth words and bits 17 to 8 of the first word are zero. A negative number is represented by the one's complement of the second word. Any real number in the range -76,000,000,000 to +76,000,000,000 can be stored as a double-precision floating-point number having a precision of 15 decimal digits.

Double-Precision Floating-Point Number (18-Bit)

```
Bit   17 16 15   14 13 12   11 10 9   8 7 6   5 4 3   2 1 0
n)    0  0  0    0  0  0    0  0  0   0 -----Exponent----
n+1)  s  --------------High Mantissa------------------
n+2)  0  --------------Mid Mantissa------------------
n+3)  0  --------------Low Mantissa------------------
```

SUBROUTINE DESCRIPTIONS

The following definitions and notation apply to the subroutine descriptions given in this section:

Notation	Meaning
AB	Hardware A and B registers
AC	Four-word software accumulator for double-precision and real numbers.
ACCZ	Four-word software accumulator for complex numbers (defined as labelled COMMON block $IMAG and where the results of all complex functions are placed)
d	A double-precision number
f	Two-word fixed-point number
i	An integer
j	A double-precision integer
r	A real number
X	Hardware X register
z	A complex number
**	Exponentiation
/	Division
$	A character commonly used as the first character of a subroutine name

The external references in table 10-1 refer to items in both tables 10-1 and 10-2. Similarly, the subroutines called in table 10-2 refer to items in both tables 10-1 and 10-2. When a subroutine has more than one name, it is indicated by multiple calls under Calling Sequence.

Table 10-1. DAS Coded Subroutines

Name	Function	Calling Sequence	External References
$HE	In A, compute i1**i2	CALL $HE,i2	$QS, $SE, $PE, $HS, $QK
$PE	In AB, compute r**i	CALL $PE,i	$SE, $QS, $QE
$QE	In AB, compute r1**r2	CALL $QE,r2	ALOG, $QM, EXP, $SE
ALOG	In AB, compute ln r. If negative, error exit with A = B = 0 and overflow = 1. If zero, set result to maximum negative number.	CALL ALOG,r	$ER, $QS, $QK, $QM, XDMU, XDAD, $FMS, $NML, $XDDI, $XDSU, $SE
EXP	In AB, compute e**r	CALL EXP,r	XDMU, $QK, $QL, $QM, $QN, $SE
ATAN	In AB, compute arctan r	CALL ATAN,r	$QM, $QL, $QN, $QK, $SE
COSINE	In AB, compute cos r	CALL COS,r	SIN, $QL, $SE
SINE	In AB, compute sin r	CALL SIN,r	$QM, XDMU, XDAD, $NML, $FMS, $SE
$SE	To transfer a list of N parameters from a calling program to a called subprogram	CALL $SE, N, LIST	None

Table 10-1. DAS Coded Subroutines *(continued)*

Name	Function	Calling Sequence	External References
SQRT	In AB, compute square root of r	CALL SQRT,r	XDDI, $FSM, $SE, XDIV
FMULDIV	In AB, compute r1*r2 with $QM, or r1/r2 with $QN. If result overflows, error exit with A = B = 0 and overflow = 1.	CALL $QM,r2 CALL $QN,r2	XDMU, $FMS, XCCI, $SE
FADDSUB	In AB, compute r1 + r2 with $QK, or r1 - r2 with $QL	CALL $QK,r2 CALL $QL,r2	$SE, $FSM, $NML, $ER
SEPMAN	Separate mantissa and characteristic of r into AB and X respectively	CALL $FMS CALL $FSM	None
FNORMAL	In AB, normalize r	CALL $NML	XDCO
XDDIV	In AB, compute f1/f2	CALL XDDI,f2	XDSU, XDCO, XDIV, XMUL
XDMULT	In AB, compute f1*f2	CALL XDMU,f2	XDAD, XDCO, XMUL
XDADD	In AB, compute f1 + f2	CALL XDAD,f2	None
XDSUB	In AB, compute f1 - f2	CALL XDSU,f2	None
XDCOMP	In AB, compute negative of f	CALL XDCO	None

Table 10-1. DAS Coded Subroutines *(continued)*

Name	Function	Calling Sequence	External References
$FLOAT	In AB, convert the i in A to floating-point and, for $QS, store result in r	CALL $PC CALL $QS,r	$SE
$IFIX	In A, convert the r in AB to i and, for $HS, store result in i	CALL $IC CALL $HS,i	$SE
IABS	In A, compute absolute i	CALL IABS,i	$SE
ABS	In AB, compute absolute r	CALL ABS,r	$SE
ISIGN	Set the sign of i1, in A, equal to that of i2	CALL ISIGN,i2	$SE
SIGN	Set the sign of r1, in AB, equal to that of r2	CALL SIGN,r2	$SE
$HN	In A, compute i1/i2	CALL $HN,i2	$SE, XDIV
$HM	In A, compute i1*i2	CALL $HM,i2	$SE, XMUL
XMUL	Software emulation of hardware multiplication	CALL XMUL,i	None
XDIV	Software emulation of hardware division	CALL XDIV,i	None

Table 10-1. DAS Coded Subroutines *(continued)*

Name	Function	Calling Sequence	External References
DSINCOS	In AC, compute sin d or cos d	CALL $DSI,d CALL $DSIN,d CALL $DCO,d CALL $DCOS,d	$STO, $DMP, SDIT, $DFR, CHEB, $SE, $DLO
DATAN	In AC, compute arctan d	CALL $DAT,d CALL $DATAN,d	$DLO, $STO, $DAD, $DSU, IF, $SE, AC, $DMP, $DDI, POLY
DEXP	In AC, compute exponential d	CALL $DEX,d CALL DEXP,d CALL TWOX,d	$DLO, $STO, $DMP, $DDI, $SE, AC, $DIT, CHEB, IF, $DFR
DLOG	In AC, compute ln d	CALL DLOG,d CALL $DLN,d	$DLO, $STO, $DAD, POLY, IF, $SE, AC, $DSU, $DMP, $DDI
POLY	In AC, compute double-precision polynomial with t terms, coefficient list starting at address c, and argument at address y	CALL POLY,t,c,y	$DLO, $DAD, $DMP
CHEB	In AC, compute shifted Chebyshev polynomial series with t + 1 terms and coefficient list starting at address c	CALL CHEB,t,c	$DLO, $STO, $DAD, $DSU, $DMP

Table 10-1. DAS Coded Subroutines *(continued)*

Name	Function	Calling Sequence	External References
DSQRT	In AC, compute square root of d	CALL $DSQ,d CALL DSQR,d	$DLO, $STO, $DNO, $DAD, $DMP, $DDI, $SE, AC
$DFR	In AC, compute fractional part of d	CALL $DFR,d	$DLO, $DNO, $DSU, $DIT, AC, $SE
IDINT	In AC, compute integral part of d	CALL $DIT,d CALL IDIN,d	$DNO, $SE
DMULT	In AC, compute d1*d2	CALL $DMP,d2 CALL $ZM,d2	$DLO, $STO, $DNO, $DAD, AC, $SE
DIVIDE	In AC, compute d1/d2	CALL $DDI,d2 CALL $ZN,d2	$DLO, $STO, $DNO, $DSU, AC, $SE
DADDSUB	In AC, compute d1 + d2 with $DAD, or d1 - d2 with $DSU	CALL $DAD,d2 CALL $DSU,d2 CALL $ZK,d2 CALL $ZL,d2	$STO, $DLO, $DNO, AC
DNORMAL	In AC, normalize d	CALL $DNO	$SE
DLOADAC	Load AC with d	CALL $DLO,d CALL $ZF,d	AC, $SE
DSTOREAC	Store AC in d	CALL $STO,d CALL $ZS,d	AC, $SE
RLOADAC	Load A with double-precision mantissa sign word from AC	CALL $ZI	AC

Table 10-1. DAS Coded Subroutines *(continued)*

Name	Function	Calling Sequence	External References
SINGLE	In AB, convert the d in AC to r	CALL $RC	AC
DOUBLE	In AC, convert the r in AB to d	CALL $YC	AC
DBLECOMP	In AC, compute negative of the d in AC	CALL $ZC	AC
$3S	Store AB in memory address m	CALL $3S,m	$SE
SNAP	Print the contents of core on logical unit (LO) (A) = starting address (B) = ending address	CALL SNAP	IOCS
A2MT 620 only	Translate in memory a character string of length n starting at s and ending at e from eight-bit ASCII to six-bit magnetic tape BCD code	CALL A2MT,n,s,e	None
MT2A 620 only	Translate in memory a character string of length n starting at s and ending at e from six-bit magnetic tape BCD code to eight-bit ASCII	CALL MT2A,n,s,e	None
DEBUG	Provide an execution address for the DEBUG package	CALL DEBUG	DBG$

Table 10-2. FORTRAN IV Coded Subroutines

Name	Function	Calling Sequence	External References
$9E	Compute ACCZ**i	CALL $9E(i)	$SE, IABS, $8F, $8M, $8N
CCOS	In ACCZ, compute cos z	CALL CCOS(z)	$SE, CSIN, $8F, $8K, $8S
CSIN	In ACCZ, compute sin z	CALL CSIN(z)	$SE, EXP, $QN, SIN, $QK, $QM, COS, $QL, $8F
CLOG	In ACCZ, compute ln z	CALL CLOG(z)	$SE, ALOG, $QM, $QK, $QN, ATAN2, $8F
CEXP	In ACCZ, compute exponential z	CALL CEXP(z)	$SE, EXP, COS, SQM, SIN, $8F

Table 10-2. FORTRAN IV Coded Subroutines *(continued)*

Name	Function	Calling Sequence	External References
CSQRT	In ACCZ, compute square root of z	CALL CSQRT(z)	$SE, SQRT, CABS, $QK, $QN, $8F
CABS	In AB, compute absolute z	CALL CABS(z)	$SE, SQRT, $QM, $QK
CONJG	In ACCZ, compute conjugate of z	CALL CONJG(z)	$SE, $8F
$AK	Add r to real part of ACCZ	CALL $AK(r)	$SE, $8S, $QK, $8F
$AL	Subtract r from the real part of ACCZ	CALL $AL(r)	$SE, $8S, $QL, $8F
$AM	Multiply ACCZ by r	CALL $AM(r)	$SE, $8S, $QM, $8F
$AN	Divide ACCZ by r	CALL $AN(r)	$SE, $8S, $QM, $8F
$AC	Convert AC to z and store in ACCZ	CALL $AC	$3S, CMPLX
CMPLX	Load ACCZ with a value having a real part r1 and an imaginary part r2	CALL CMPLX(r1,r2)	$SE, $8F
$8K	Add z to ACCZ	CALL $8K(z)	$SE, $8S, $QK, $8F
$8L	Subtract z from ACCZ	CALL $8L(z)	$SE, $8S, $QL, $8F
$8M	Multiply ACCZ by z	CALL $8M(z)	$SE, $8S, $QM, $QL, $QK, $8F
$8N	Divide ACCZ by z	CALL $8N(z)	$SE, $8S, $QM, $QK, $QN, $QL, $8F

Table 10-2. FORTRAN IV Coded Subroutines *(continued)*

Name	Function	Calling Sequence	External References
$ZD	Compute negative of z	CALL $ZD	$8S, $8F
AIMAG	Load AB with the imaginary part of z	CALL AIMAG(z)	$SE
$OC	Load AB with the real part of ACCZ	CALL $OC	$8S
REAL	Load AB with the real part of z	CALL REAL(z)	$SE
$8F	Load ACCZ with z	CALL $8F(z)	$SE
$8S	Store ACCZ in z	CALL $8S(z)	$SE, $3S
$XE	Compute d**i where d is in AC	CALL $XE(i)	$SE, $ZF, MOD, $ZM, $HN, $ZN
$YE	Compute d**r where d is in AC	CALL $YE(r)	$SE, $ZS, DBLE, $ZE, $ZF
$ZE	Compute d1**d2 where d1 is in AC	CALL $ZE(d2)	$SE, $ZS, DEXP, DLOG, $ZM
DATAN2	In AC, compute arctan (d1/d2)	CALL DATAN2(d1,d2)	$SE, $ZF, $ZS, $ZI, $ER, $ZN, $ZL, $ZK, DATAN
DLOG10	In AC, compute log d	CALL DLOG10(d)	$SE, DLOG, $ZM
DMOD	In AC, compute d1 modulo d2	CALL DMOD(d1,d2)	$SE, DINT, $ZF, $ZN, $ZS, $ZM, $ZL, $ZC
DINT	In AC, compute integer portion of d	CALL DINT(d)	$SE, $ZF, $JC, $XC
DABS	In AC, compute absolute d	CALL DABS(d)	$SE, $ZF, $ZI,

Table 10-2. FORTRAN IV Coded Subroutines *(continued)*

Name	Function	Calling Sequence	External References
			$ZN
DMAX1	In AC, select the maximum value in the set d1, d2,...,dn	CALL DMAX1(d1,d2, ...,dn,0)	$SE, $ZF, ZS, IFA, $ZL, $ZI
DMIN1	In AC, select the minimum value in the set d1, d2, ...,dn	CALL DMIN1(d1,d2, ...,dn,0)	$SE, $ZF, ZS, IFA, $ZL, $ZI
DSIGN	Set the sign of d1 equal to that of d2	CALL DSIGN (d1,d2)	$SE, $ZF, $ZI, $ZN
$YK	Add r to AC	CALL $YK(r)	$SE, $ZS, DBLE, $ZK
$YL	Subtract r from AC	CALL $YL(r)	$SE, $ZS, DBLE, $ZL, $ZC
$YM	Multiply AC by r	CALL $YM(r)	$SE, $ZS, DBLE, $ZM
$YN	Divide AC by r	CALL $YN(r)	$SE, $ZS, DBLE, $ZF, $ZN
DBLE	In AC, convert r to d	CALL DBLE(r)	$SE, $YC
$XC	In AC, convert i to d where i is in A	CALL $XC	$PC, $YC
TANH	In AB, compute tanh r	CALL TANH(r)	$SE, $QK, EXP, $QL, $QN
ATAN2	In AB, compute arctan (r1/r2)	CALL ATAN2(r1,r2)	$SE. $ER, ATAN, $QK. $QL, $QN
ALOG10	In AB, compute log r	CALL ALOG10(r)	$SE, ALOG, $QM
AMOD	In AB, compute r1 modulo r2	CALL AMOD(r1,r2)	$SE, AINT, $QN,

Table 10-2. FORTRAN IV Coded Subroutines *(continued)*

Name	Function	Calling Sequence	External References
			$QM, $QL
AINT	In AB, truncate r	CALL AINT(r)	$SE, $IC, $PC
AMAX1	In AB, select the maximum value in the set r1,r2,..., rn	CALL AMAX1(r1,r2 ...,rn,0)	SE, IFA, $QL
AMIN1	In AB, select the minimum value in the set r1,r2,..., rn	CALL AMIN1(r1,r2 ...,rn,0)	SE, IFA, $QL
AMAX0	In AB, select the maximum value in the set i1,i2,..., in and convert to r	CALL AMAX0(i1,i2, ...,in,0)	SE, IFA, FLOAT
AMIN0	In AB, select the minimum value in the set i1,i2,..., in and convert to r	CALL AMIN0(i1,i2, ...,in,0)	SE, IFA, FLOAT
DIM	In AB, compute the positive difference between r1 and r2	CALL DIM(r1,r2)	$SE, $QL
FLOAT	In AB, convert i to r	CALL FLOAT(i)	$SE, $PC
SNGL	In AB, convert d to r	CALL SNGL(d)	$SE, $ZF, $RC
MAX0	In A, select the maximum value in the set i1,i2, ...,in	CALL MAX0(i1,i2, ...,in,0)	SE, IFA
MIN0	In A, select the minimum value in the set i1,i2 ...,in	CALL MIN0(i1,i2, ...,in,0)	SE, IFA
MAX1	In A, select the maximum value in the set r1,r2, ...,rn and convert to i	CALL MAX1(r1,r2, ...,rn,0)	SE, IFA, $QL, IFIX
MIN1	In A, select the minimum	CALL MIN1(r1,r2,	SE, IFA, $QL,

Table 10-2. FORTRAN IV Coded Subroutines *(continued)*

Name	Function	Calling Sequence	External References
	value in the set r1,r2, ...,rn and convert to i	...,rn,0)	IFIX
MOD	In A, compute i1 modulo i2	CALL MOD(i1,i2)	$SE, $HN, $HM
INT	In A, truncate r and convert to i	CALL INT(r)	$SE, $IC
IDIM	In A, compute the positive difference between i1 and i2	CALL IDIM(i1,i2)	$SE
IFIX	In A, convert r to i	CALL IFIX(r)	$SE, $IC
$JC	In AC, convert d to i and store result in A	CALL $JC	$RC, $IC

SECTION 11 - MOS OPERATING PROCEDURES

The installation system library (ISL) of the MOS is available on punched paper tape, or, for systems having 12K of memory, on magnetic tape or punched cards. From the ISL, the user performs a system preparation to obtain a system file (Section 8). The following procedures assume the presence of a system file and deal only with bootstrapping and initialization of MOS.

DEVICE INITIALIZATION

CARD READER

a. Turn on the card reader.

b. Place two blank cards after the last control-directive card of the input deck.

c. Place the input deck in the card hopper.

d. Press CLEAR and START.

CARD PUNCH

a. Turn on the card punch.

b. Place blank cards in the card hopper.

c. Press START.

33/35 ASR TELETYPE

a. Turn on the Teletype.

b. Set the Teletype in off-line mode and simultaneously press the CONTROL and D, then the CONTROL and T, and finally CONTROL and Q keys.

c. Set the Teletype on-line.

HIGH-SPEED PAPER TAPE READER

a. Turn on the paper tape reader.

b. Position the input paper tape in the reader with blank leader at the reading station and close the reading gate.

c. Set the LOAD/RUN switch to RUN.

MAGNETIC TAPE UNIT

a. Turn on the magnetic tape unit.

b. Mount the input magnetic tape.

c. Position the magnetic tape to the loading point.

d. Ready the magnetic tape unit so it can be used by the computer.

FIXED-HEAD DISC UNIT *(Model 620-38C and 620-43B, C, D)*

a. Press the AC POWER switch.

b. Wait for the AC POWER indicator to light.

c. Press the DC POWER switch.

MOVING-HEAD DISC UNIT *(Model 620-36, -37)*

a. Turn on the disc unit.

b. Set the START/STOP switch to START.

c. Wait for the disc unit to reach operating speed (READY indicator lights).

d. Turn off WRITE PROTECT.

MOVING-HEAD DISC UNIT *(Models 620-35)*

a. Turn on the disc unit.

b. Wait for the disc unit to reach operating speed (DISC READY indicator lights).

BOOTSTRAP

To enter the bootstrap loading routine (table 11-1) into computer memory, follow the bootstrap loading procedures given in section 8.

To initiate the bootstrap, reset the A, B, X, P, and instruction registers. Then, press SYSTEM RESET and RUN (for V73 and 620/f press RESET, position STEP/RUN, and press START).

Table 11-1. MOS System File Bootstrap Routines

Address	Magnetic Tape Unit	620-46 to -49 Drum	620-38C Fixed-Head Disc	620-35 Moving-Head Disc	620-39 Moving-Head Disc	620-40, -41 Moving-Head Disc	620-37 Moving-Head Disc
00000	104110	1000yy	1000yy	005006	005003	005006	005003
00001	101210	006020	006020	010024	101316	010024	100416
00002	000005	000012	000012	140034	000013	140034	100216
00003	001000	010014	010014	001002	100716	001002	103116
00004	000001	1031xx	1031xx	001001	100216	001001	101016
00005	030016	006120	006120	120034	103016	120034	000010
00006	100010	000350	000350	100015	000035	100015	001000
00007	102510	1031yy	1031yy	1000yy	101516	1000yy	000004
00010	055000	1000xx	1000xx	1031xx	000025	1031xx	100416
00011	005144	100014	100014	006120	001000	006120	100316
00012	101110	103214	103214	000121	000007	000121	103116
00013	000007	1010xx	1010xx	1031yy	010033	1031yy	100021
00014	101210	001001	001001	1000xx	100716	1000xx	014011
00015	001001	001000	001000	103215	100416	103215	103120
00016	001000	000013	000013	001040	103216	001040	124011
00017	000012			000026	1000yy	000026	103121
00020				100415	1031xx	100415	100020
00021				005122	120036	005122	100016
00022				101415	1031yy	101415	101416
00023				000002	1000xx	000002	000022
00024				001000	100016	001000	101516
00025				000022	101216	000022	000000
00026				100515	000025	100115	001000
00027				101015	101016	101015	001001
00030				000001	000027	000001	000550
00031				005144	001010	005144	000000
00032				001000	000000	001000	
00033				000027	001000	000027	
00034				001520	001001	001520	
00035					000312		
00036					000550		

Where xx = even BIC address and yy = odd BIC address.

SYSTEM (RE)INITIALIZATION

The executive component of MOS (re)initializes the system:

a. At bootstrap time

b. When /JOB is read

c. When /ENDJOB is read

d. When the operator resets the A, B, X, P, and instruction registers, presses the SYSTEM RESET, and presses RUN (for the V73 and 620/f, the operator presses RESET, positions STEP/RUN to RUN, and presses START).

Upon (re)initialization:

a. All logical unit assignments are set to their default values.

b. System flags denoting errors, processor options, and loader options are reset.

To (re)execute the user's program in memory:

a. Set the P register to 000002 and the instruction register to zero.

b. Press SYSTEM RESET (RESET for the V73 and 620/f).

c. Press RUN (for the V73 and 620/f, position STEP/RUN to RUN and press START).

To enter the dump program prior to (re)initialization through the executive:

a. Set the P register to 000004 and the instruction register to zero.

b. Press SYSTEM RESET (RESET for the V73 and 620/f).

c. Press RUN (for the V73 and 620/f, position STEP/RUN to RUN and press START).

SECTION 12 - USER-CODED I/O DRIVERS

MOS permits augmenting the system with I/O drivers coded by the user. To develop an I/O routine and place it in MOS:

a. Code the driver according to the applicable I/O device specification.

b. Assemble the driver using the DAS MR assembler.

c. Add the driver to the system file using the system preparation program.

d. Use the driver through I/O control calls.

DEVICE SPECIFICATION TABLE

When a driver controls a single logical unit, the first 16 or more words comprise the device specification table (DST). When multiple units are controlled, such as magnetic tape units, a DST is required for each. Each DST is a separate assembly linked to its driver through externals. The DST:

a. Transfers parameters of the user I/O request to the driver

b. Determines if an I/O driver can accommodate an I/O request

c. Obtains the results of I/O function requests

Table 12-1 lists I/O driver entry names and their corresponding device names and device addresses as recognized by MOS Input/Output Control. The device addresses are omitted for those peripheral devices which are presently not supported by VDM. If a user replaces an existing driver or incorporates a driver for one of the devices which is not supported, it is suggested that the listed entry names, device names, and device addresses be retained.

DST words have the following significance:

Word	Meaning
0	Actual number of transfers
1	I/O status
2-3	I/O driver name
4	I/O request flag and operation code
5	Count parameter
6	Address parameter
7	Address of I/O checking routine
8	Address of I/O checking routine
9	Address of reading routine
10	Address of writing routine
11	Address of write-end-of-file routine
12	Address of rewind routine
13	Address of skip-files routine
14	Address of skip-records routine
15	Address of function routine
16-n	Defined by driver

WORD 0

Modified by:	I/O driver
Used by:	User programs when information on the number of transfers is required

Word 0 is associated with the entry name of the I/O driver since all remaining DST words are referenced relative to word 0. I/O driver entry names comprise three characters, the first of which is the dollar sign ($); the second, a number (0 through 9); and the third, a letter or number.

Word 0 contains the number of words transferred for an I/O reading or writing request, or the number of files or records remaining to be skipped for a skip request. The I/O driver puts this information into word 0 of the DST.

WORD 1

Modified by:	I/O driver
Used by:	I/O control and user programs

Bits 0 through 2 of word 1 are examined by I/O control during a status request call to determine the appropriate return as follows:

Value	Meaning
0	Normal return
1	Error
2	End of file
3	End or beginning of device
4	Last operation not complete (busy)

Bits 3 through 15 (17 on 18-bit computers) are not examined by I/O control. However, for uniformity in the I/O driver, the following meanings are ascribed to the bit positions:

Bit	Meaning
3-5	Temporary storage for I/O driver
6	Last operation was rewind
7	Odd-length record detected
8	Error detected

Bit	Meaning
9	Unit ready
10	Unit rewinding
11	End of file detected
12	End of device detected
13	Beginning of device detected
14	Last operation ignored
15	Status not valid
16	Not used (18-bit computers only)
17	Status not valid (18-bit computers only)

WORDS 2 and 3

Modified by: Not modified
Used by: I/O control and user programs

Words 2 and 3 contain the four-character ASCII name of the peripheral device for the I/O driver.

WORD 4

Modified by: I/O control and I/O driver
Used by: I/O control and I/O driver

When word 4 is positive, I/O control is advised that an I/O request is in process or pending and a new I/O request must wait. When word 4 is negative, I/O control is notified that the I/O driver is available. When an I/O request is made for an available I/O driver, I/O control puts the function code from the user's I/O request in bits 7 through 0 of word 4, making it positive.

WORD 5

Modified by: I/O control and I/O driver
Used by: I/O driver

When an I/O request is made for an I/O driver and no other request is pending (word 4 negative), I/O control puts the count (if greater than zero) from the user's I/O call in word 5.

WORD 6

Modified by: I/O control and I/O driver
Used by: I/O driver

When an I/O request is made for an I/O driver and no other request is pending (word 4 negative), I/O control puts the data address from the user's I/O call in word 6. The most significant bit is always zero.

WORDS 7 AND 8

Modified by:	Not modified
Used by:	I/O control

Words 7 and 8, which contain the same address, are used as follows:

a. Before I/O control passes to the I/O driver to perform the requested I/O function, it transfers to the I/O driver by a Jump and Mark Indirect (JMPM*) instruction to the address of word 7. Then, the I/O driver determines if the peripheral device is available. If so, it sets a positive condition code in the A register, permitting I/O control to enter the I/O driver a second time to process the request. If the device is unavailable, the A register is set to zero, signaling I/O control not to make an I/O request at this time.

b. Every time a call requesting status is made to I/O control, it checks the I/O driver request flag (most significant bit of word 4). If no request is in process, control passes to the I/O driver with a JMPM* to the address given in word 7. The I/O driver sets the condition code in the A register before returning control to I/O control.

WORD 9

Modified by:	Not modified
Used by:	I/O control

If the I/O driver can read, word 9 contains the address used by I/O control when reading is requested of this I/O driver. If the I/O driver cannot read, word 9 is zero.

To read, I/O control passes to the I/O driver with the X register pointing to word 0 of the I/O driver DST. When the reading is complete, the I/O driver returns control to I/O control with a Jump Indirect (JMP*) instruction to the address in word 7.

The I/O driver sets a condition code in the A register before returning to I/O control. A negative A register indicates that more than 500 microseconds were spent in the I/O driver, and the physical unit table ($PUT, section 2) is scanned again because another I/O operation may have finished in the interim. A zero in the A register indicates that the I/O driver is busy. A positive A register (nonzero) indicates that the I/O driver is free.

WORD 10

Modified by:	Not modified
Used by:	I/O control

If the I/O driver can write, word 10 contains the address used by I/O control when writing is requested of this I/O driver. If the I/O driver cannot write, word 10 is zero.

To write, I/O control passes to the I/O driver with the X register pointing to word 0 of the I/O driver DST. When writing is complete, the I/O driver returns control to I/O control with a JMP* to the address in word 7.

The I/O driver sets a condition code in the A register before returning to I/O control. A negative A register indicates that more than 500 microseconds were spent in the I/O driver, and the PUT is scanned again because another I/O operation may have finished in the interim. A zero in the A register indicates that the I/O driver is busy. A positive A register (nonzero) indicates that the I/O driver is free.

WORD 11

Modified by:	Not modified
Used by:	I/O control

If the I/O driver can write an end of file, word 11 contains the address used by I/O control when a write-end-of-file is requested of this I/O driver. If the I/O driver cannot write an end of file, word 11 is zero.

To write an end of file, I/O control passes to the I/O driver with the X register pointing to word 0 of the I/O driver DST. When the writing is complete, the I/O driver returns control to I/O control with a JMP* to the address in word 7.

The I/O driver sets a condition code in the A register before returning to I/O control. A negative A register indicates that more than 500 microseconds were spent in the I/O driver, and the PUT is scanned again because another I/O operation may have finished in the interim. A zero in the A register indicates that the I/O driver is busy. A positive A register (nonzero) indicates that the I/O driver is free.

WORD 12

Modified by:	Not modified
Used by:	I/O control

If the I/O driver can rewind, word 12 contains the address used by I/O control when rewinding is requested of this I/O driver. If the I/O driver cannot rewind, word 12 is zero.

To rewind, I/O control passes to the I/O driver with the X register pointing to word 0 of the I/O driver DST. When rewinding is complete, the I/O driver returns control to I/O control with a JMP* to the address in word 7.

The I/O driver sets a condition code in the A register before returning to I/O control. A negative A register indicates that more than 500 microseconds were spent in the I/O driver, and the PUT is scanned again because another I/O operation may have finished in the interim. A zero in the A register indicates that the I/O driver is busy. A positive A register (nonzero) indicates that the I/O driver is free.

WORD 13

Modified by: Not modified
Used by: I/O control

If the I/O driver can skip files, word 13 contains the address used by I/O control when file-skipping is requested of this I/O driver. If the I/O driver cannot skip files, word 13 is zero.

To skip files, I/O control passes to the I/O driver with the X register pointing to word 0 of the I/O driver DST. When the file-skipping is complete, the I/O driver returns control to I/O control with a JMP* to the address in word 7.

The I/O driver sets a condition code in the A register before returning to I/O control. A negative A register indicates that more than 500 microseconds were spent in the I/O driver, and the PUT is scanned again because another I/O operation may have finished in the interim. A zero in the A register indicates that the I/O driver is busy. A positive A register (nonzero) indicates that the I/O driver is free.

WORD 14

Modified by: Not modified
Used by: I/O control

If the I/O driver can skip records, word 14 contains the address used by the I/O control when record-skipping is requested of this I/O driver. If the I/O driver cannot perform a skip record function, word 14 is zero.

To skip records, I/O control passes to the I/O driver with the X register pointing to word 0 of the I/O driver DST. When the record-skipping is complete, the I/O driver returns control to I/O control with a JMP* to the address in word 7.

The I/O driver sets a condition code in the A register before returning to I/O control. A negative A register indicates that more than 500 microseconds were spent in the I/O driver, and the PUT is scanned again because another I/O operation may have finished in the interim. A zero in the A register indicates that the I/O driver is busy. A positive A register (nonzero) indicates that the I/O driver is free.

WORD 15

Modified by: Not modified
Used by: I/O control

If the I/O driver can perform a function not otherwise covered, word 15 contains the address used by I/O control when a function is requested of this I/O driver. If the I/O driver cannot perform such a function, word 15 is zero.

To perform the function, I/O control passes to the I/O driver with the X register pointing to word 0 of the I/O driver DST. When the function is complete, the I/O driver returns control to I/O control with a JMP* to the address in word 7.

The I/O driver sets a condition code in the A register before returning to I/O control. A negative A register indicates that more than 500 microseconds were spent in the I/O driver, and the PUT is scanned again because another I/O operation may have finished in the interim. A zero in the A register indicates that the I/O driver is busy. A positive A register (nonzero) indicates that the I/O driver is free.

Table 12-1. I/O Drivers and Peripheral Devices

Peripheral Device	Device Name	Entry Name	Device Address
Magnetic tape (controller 0, unit 0)	MT00	$00	010
Magnetic tape (controller 0, unit 1)	MT01	$01	010
Magnetic tape (controller 0, unit 2)	MT02	$02	010
Magnetic tape (controller 0, unit 3)	MT03	$03	010
Magnetic tape (controller 1, unit 0)	MT10	$04	011
Magnetic tape (controller 1, unit 1)	MT11	$05	011
Magnetic tape (controller 1, unit 2)	MT12	$06	011
Magnetic tape (controller 1, unit 3)	MT13	$07	011
Magnetic tape (controller 2, unit 0)	MT20	$08	012
Magnetic tape (controller 2, unit 1)	MT21	$09	012
Magnetic tape (controller 2, unit 2)	MT22	$0A	012
Magnetic tape (controller 2, unit 3)	MT23	$0B	012

Table 12-1. I/O Drivers and Peripheral Devices *(continued)*

Peripheral Device	Device Name	Entry Name	Device Address
Magnetic tape (controller 3, unit 0)	MT30	$0C	013
Magnetic tape (controller 3, unit 1)	MT31	$0D	013
Magnetic tape (controller 3, unit 2)	MT32	$0E	013
Magnetic tape (controller 3, unit 3)	MT33	$0F	013
Teletype keyboard/ printer 1 (controller 0, unit 0)	TY00	$0G	001
Teletype paper tape punch 1 (controller 0, unit 0)	TP00	$0H	001
Teletype paper tape reader 1 (controller 0, unit 0)	TR00	$0I	001
Teletype keyboard/ printer 2 (controller 1, unit 0)	TY10	$0J	---
Teletype paper tape punch 2 (controller 1, unit 0)	TP10	$0K	---
Teletype paper tape reader 2 (controller 1, unit 0)	TR10	$0L	---

Table 12-1. I/O Drivers and Peripheral Devices *(continued)*

Peripheral Device	Device Name	Entry Name	Device Address
Card reader 1	CR00	$0M	030
Card reader 2	CR10	$0N	---
High-speed paper tape reader/punch 1	PT00	$0O	037
High-speed paper tape reader/punch 2 (unformatted)	PT10	$0P	037
Line printer 1	LP00	$0Q	035
Line printer 2	LP10	$0R	---
Card punch 1	CP00	$0S	031
Card punch 2	CP10	$0T	---
Drum	DR00	$10	014
(controller 0, unit 0)	•	•	•
•	•	•	•
•	•	•	•
Drum	DR09	$19	014
(controller 0, unit 0)			
Disc	DK00	$10	015
(controller 0, unit 0)	•	•	(620-40/41)
•	•	•	016
•	•	•	(620-39)
Disc	DK09	$19	
(controller 0, unit 0)			
Disc	DK10	$20	015
(controller 0, unit 1)	•	•	(620-40/41)
•	•	•	016
•	•	•	(620-39)
Disc	DK19	$29	
(controller 0, unit 1)			

Table 12-1. I/O Drivers and Peripheral Devices *(continued)*

Peripheral Device	Device Name	Entry Name	Device Address
Disc	DK40	$50	016
(controller 1, unit 0)	.	.	(620-40/41)
	.	.	017
	.	.	(620-39)
Disc	DK49	$59	
(controller 1, unit 0)			
First buffer interlace controller			020-021
Second buffer interlace controller			022-023
Third buffer interlace controller			024-025
Fourth buffer interlace controller			026-027

I/O DRIVER PROGRAMMING EXAMPLES

Example 1

Output an alphanumeric character string to the Teletype:

```
            NAME        $TY
*           DST
$TY         DATA        0           WORDS TRANSFERRED
            DATA        0           I/O STATUS
            DATA        'TY99'      I/O DRIVER NAME
            DATA        -1      -   I/O FLAG AND OP CODE
            DATA        0           COUNT PARAMETER
            DATA        0           LOCATION PARAMETER
            DATA        $TCK        CHECK I/O ADDRESS
            DATA        $TCK        CHECK I/O ADDRESS
            DATA        0           UNUSED
            DATA        $TWR        ADDRESS OF WRITE I/O
            DATA        0           UNUSED
            DATA        0           UNUSED
            DATA        0           UNUSED
            DATA        0           UNUSED
            DATA        0           UNUSED
*
$TCK        DATA        0           TTY AVAILABILITY UNDE-
                                    TERMINED,ASSUME READY
            INCR        1
            JMP*        $TCK
*
$TWR        LDA         5,1         GET COUNT PARAMETER
            STA         0,1         STORE IT IN WORD 0
            LDB         6,1         GET DATA LOCATION
            EXC         0401        INITIALIZE TTY
            LDAI        0201
            JMPM        $TOAR       OUTPUT PRINT ENABLE
$TWR1       LDA         5.1         GET COUNT
            DAR
            JAN         $TWR2       JMP IF END OF WORD
```

```
            STA         5,1
            LDA         0,2         GET DATA WORD
            LRLA        8
            JMPM        $TOAR       OUTPUT LEFT CHARACTER
            LRLA        8
            JMPM        $TOAR       OUTPUT RIGHT CHARACTER
            IBR
            JMP         $TWR1
$TWR2       LDAI        0215
            JMPM        $TOAR       OUTPUT CARRIAGE RETURN
            LDAI        0212
            JMPM        $TOAR       OUTPUT LINE FEED
            LDAI        0204
            JMPM        $TOAR       OUTPUT PRINT OFF CHAR
            DECR        1           A REGISTER NEGATIVE
            STA         4,1         TURN OFF REQUEST FLAG
*           JMP*        $TCK
$TOAR       DATA        0
            SEN         0101,*+4    WRITE REGISTER READY
            JMP         *-2         NOT READY, WAIT
            OAR         1           OUTPUT A CHARACTER
            JMP*        $TOAR       RETURN
            END
```

Example 2

Read binary records from magnetic tape unit 0 on controller 0, device address 010.

```
            NAME        $MT
*           DST
$MT         DATA        0
            DATA        0
            DATA        'MT99'
            DATA        -1
            DATA        0
            DATA        0
            DATA        $MTS        CHECK I/O ADDRESS
            DATA        $MTS        CHECK I/O ADDRESS
            DATA        $MRD        ADDRESS OF READ I/O
            DATA        0
            DATA        0
            DATA        0
            DATA        0
            DATA        0
*           STATUS ROUTINE
$MTS        DATA        0
            SEN         0210,A      IF MTU READY
```

```
            TZA                     SET (AR) = 0
            JMP*        $MTS        RETURN (MTU BUSY)
A           LDA         1,1         GET WORD 1 OF DST
            JAN         B           IF STATUS INVALID
            INCR        01          SET A = 1
            JMP*        $MTS        RETURN (MTU READY)
B           DECR        01          SET A = -1
            STA         4,1         SET WORD 4 OF DST TO
                                    'NOT BUSY'
            SEN         010,ERR     IF ERROR OCCURRED
            SEN         0310,EOF    IF END OF FILE READ
            SEN         0510,EOT    IF END OF TAPE FOUND
            SEN         0610,BOT    IF BEGINNING OF TAPE
            JMP         NORM        INDICATE NORMAL STATUS
BOT         BSS         0
EOT         IAR                     SET WORD 1 OF DST TO 3
EOF         IAR                     SET WORD 1 OF DST TO 2
ERROR       IAR                     SET WORD 1 OF DST TO 1
NORM        IAR                     SET WORD 1 OF DST TO 0
            STA         1,1
            JMP*        $MTS        RETURN, STATUS SET
*           READ ROUTINE
$MRD        LDB         6,1         GET START MEMORY ADDR
            EXC         010         START MTU
C           SEN         0110,D      IF WORD READY
            SEN         0210,E      IF MTU STOPPED
            JMP         C           WAIT
D           CIA         010         INPUT WORD
            STA         0,2         STORE WORD
            IBR                     BUMP MEMORY POINTER
            INR         0,1         INCREMENT NO OF XFERS
            LDA         5,1
            DAR
            STA         5,1         DECREMENT COUNT
            JAZ         E           IF FINISHED
            JMP         C           CONTINUE
E           LDAI        0100004
            STA         1,1         STATUS INVALID, MTU BUSY
            TZA                     (AR) = 0 FOR MTU BUSY
            JMP*        $MTS        RETURN
            END
```

I/O SUPPORT SUBROUTINES

To prevent possible data loss when several I/O transfers are operating concurrently, MOS has a subroutine, IOOK, for determining if enough processor time is available before initiating an I/O operation. This subroutine is used by the I/O driver prior to data transfer. The subroutine calling sequence is:

```
JMPM        IOOK
```

Upon IOOK entry, the A register contains the I/O algorithm factor for that I/O device. The I/O algorithm factor depends on whether the device is operating in BIC or programmed data transfer mode. The equations for the two types of transfers are:

$$Fb = (Tb/Tt) + 0.1(Tb/Tt)$$

$$Fp = (N * C)/Tt + 0.1((N * C)/Tt)$$

where

Fb = I/O algorithm factor for BIC transfers
Fp = I/O algorithm factor for programmed transfers
C = CPU memory cycle time (microseconds)
Tt = maximum transfer rate of I/O device (microseconds/word)
Tb = maximum transfer rate of BIC (microseconds/word)
N = number of CPU memory cycles required by data transfer

On returning from IOOK, the I/O driver examines the A register. If the A register is positive, the I/O operation can be performed. If the A register is negative, more time is needed for the I/O operation than is available. In the latter case, the new I/O operation cannot start and the driver exits to I/O control indicating a busy device (A = 0).

Upon completion of a successful I/O operation, the I/O driver removes the algorithm factor from the system timing variable by again calling IOOK with the two's complement of the I/O algorithm factor in the A register.

Example

If device A is operating with a BIC and has a data transfer rate of one word every 9.9 microseconds, and device B is operating with another BIC and has a data transfer rate of one word every 19.8 microseconds, can device C be operated over the I/O bus if it requires the following programmed transfers:

```
$1      SEN     DEVC,$2     DEVICE READY
        JMP     *-2         NO, WAIT
$2      CIA     DEVC        YES, GET DATA
        STA     0,1         STORE DATA IN ADDR(X)
```

```
INCR        045         INCREMENT DATA ADDRESS
SUB         EADD
JAN         $1          JMP IF NOT END ADDRESS
```

The I/O algorithm factor for device A is: (4.95/9.9) + 0.1(4.95/9.9) = 0.55. The I/O algorithm factor for device B is: (4.95/19.8) + 0.1(4.95/19.8) = 0.275. The I/O algorithm factor for device C is: (11.25 * 1.8)/81 + 0.1 ((11.25 * 1.8)/81) = 0.275. The sum of the I/O algorithm factors for devices A and B is 0.825, which is less than one. Thus, A and B use only about 82.5 percent of the available memory cycles. However, if C were added, the sum would be 1.1, requiring more memory cycles than available (110 percent). Therefore, C cannot be activated at this time and must wait for either A or B to complete its operation.

I/O STATUS MESSAGES

The operator alert subroutine is called by the I/O driver to advise the operator that a peripheral device is not available. The calling sequence is:

```
JMPM        IOA$
DATA        DELY
```

The X register points to word 0 of the I/O driver. DELY is the address of a two-memory-word data block in the I/O driver.

IOA$ examines and increments the first word of the DELY data block and returns to the calling routine if this word is negative. If the first word is positive, IOA$ copies the contents of the second word into the first word and moves the name of the peripheral device from words 2 and 3 of the I/O driver DST into the message:

```
xxxx - NOT READY
```

This message is output at intervals of approximately 10 seconds to the Teletype printer by the resident message-printing subroutine. IOA$ does not save any register contents when returning to the calling routine.

BIC CONTROL

When an I/O driver is controlling a device that can operate on a BIC, subroutines BIR$ and BIA$ are used. In addition, a BIC control table is defined by the driver for each I/O device controller. A pointer to the table can be appended to the end of the DST for reference purposes. The BIC control subroutines used by the I/O driver have the same calling sequence:

```
CALL    BIR$    With the B register pointing
        BIA$    to word 0 of the control table
```

BIC CONTROL TABLE

The words in the BIC control table are defined as follows:

Word	Definition
0	Contains the BIC address associated with this device (figure 12-1). Set to -01 if none.
1	Contains the BIC initial address set and used by BIR$.
2	Contains the abnormal BIC stop flag set by BIR$. It is set after a BIC operation in which an abnormal BIC stop occurs.
3	Contains the word count (i.e., the number of words to be input or output through the BIC).
4	Contains the starting memory address for the BIC transfer.
5-n	Contains controller commands and flag information specific to each device.

The controller commands are built at assembly time with the appropriate device address (i.e., each controller has its own device address, hardwired).

The flag information is used by the driver to determine and monitor such things as the direction of a skip, the type of skip (records or files), etc.

NOTE

Both words 3 and 4 are used by the BIC routines to initiate an operation. These words should be initialized by the driver before BIA$ is called.

BIR$

BIR$ determines if there is a BIC associated with the control table by testing word 0 of the BIC control table. A negative value indicates that there is no BIC. BIR$ returns and sets the A register negative. If word 0 is positive, BIR$ uses the number to build the BIC instructions.

BIR$ determines if the BIC is busy. If the BIC is busy, BIR$ returns and sets the A register negative. If the BIC is not busy, BIR$ continues.

BIR$ determines if an abnormal BIC stop has occurred. If it has, BIR$ returns and sets the A register negative. If not, BIR$ returns and sets the A register positive.

NOTE

The contents of the X register are destroyed.

BIA$

BIA$ determines if there is a BIC associated with the control table by testing word 0 of the BIC control table. A negative value indicates that there is no BIC. BIA$ returns and sets the A register negative. If word 0 is positive, BIA$ uses the number to build the BIC instructions.

BIA$ initializes the BIC and outputs the initial address from word 4 of the BIC control table.

BIA$ adds one less than the count from word 3 of the BIC control table to the initial address to obtain the final address, and then outputs it. BIA$ activates the BIC and returns to the caller.

NOTE

The contents of the X register are destroyed.

SECTION 13 - STATUS AND ERROR MESSAGES

EXECUTIVE

During operation of the executive, the following status and error messages can be posted on the system output device:

Message	Cause	Action
ILL DIR	Invalid directive read by executive	Directive ignored. If SI = TY00, waits for input of next directive. If SI ≠ TY00, reads but ignores SI until next /JOB or /ENDJOB.
READ ERR ON SI INPUT /ASSIGN SI = TY	I/O error while reading directive	Using the TTY, manually input the directive that caused the error.
/ASSIGN SI = TY	An end of file or end of device detected during reading on SI file	System waits for operation action. The next directive should be entered through the TTY.
ASSIGN DIR PARAM ERR	Either L or R parameter missing from an /ASSIGN	Directive ignored. If SI = TY00, waits for input of next directive. If SI ≠ TY00, reads but ignores SI until next /JOB or /ENDJOB.
L = R INCOMPATIBLE	R unit cannot perform the function required by the L unit on /ASSIGN	Directive ignored. If SI = TY00, waits for input of next directive. If SI ≠ TY00, reads but ignores SI until next /JOB or /ENDJOB.

Message	Cause	Action
CANNOT ASSIGN SI = DUM	Attempt to assign system input file to a dummy physical device	Directive ignored. If SI = TY00, waits for input of next directive. If SI ≠ TY00, reads but ignores SI until next /JOB or /ENDJOB.
L.U.xxxx MAY NOT BE REASSIGN	Attempt to reassign SO or SF logical unit	Directive ignored. If SI = TY00, waits for input of next directive. If SI ≠ TY00, reads but ignores SI until next /JOB or /ENDJOB.
ILL LUN/NAME USED IN DIR	Invalid parameter on /IOLIST	List terminated. If SI = TY00, waits for input of next directive. If SI ≠ TY00, reads but ignores SI until next /JOB or /ENDJOB.
LUN.xx END/BEG OF DEV ERR	Physical end or beginning of device encountered on unit xx prior to completion of certain executive functions	Directive ignored. If SI = TY00, waits for input of next directive. If SI ≠ TY00, reads but ignores SI until next /JOB or /ENDJOB.
CPY INPT UNIT UNABL TO READ	Invalid /COPYA or /COPYB (L unit cannot be read)	Directive ignored. If SI = TY00, waits for input of next directive. If SI ≠ TY00, reads but ignores SI until next /JOB or /ENDJOB.
CPY OUTPT UNIT UNABL TO WRTE	Invalid /COPYA or /COPYB (R unit cannot be written on)	Directive ignored. If SI = TY00, waits for input of next directive. If SI ≠ TY00, reads but ignores SI until next /JOB or /ENDJOB.
CPY READ ERR	I/O reading error during copying	Copying terminated. Read next directive from SI.

CPY WRTE ERR	I/O writing error during copying	Copying terminated. Read next directive from SI.
END OF CPY OUTPT DEV	Physical end of device encountered during copying	Copying terminated. Read next directive from SI.
ILL LOAD/ULOAD DIR PARAM xx	Undefined parameters encountered during processing of /LOAD or /ULOAD, where xx = first two characters of invalid parameter	Directive ignored. If SI = TY00, waits for input of next directive. If SI ≠ TY00, reads but ignores SI until next /JOB or /ENDJOB.
xx = N INVALID	One or more of the following parameters out of range on /LOAD or /ULOAD: $0 \leq RP \leq$ core size $0 \leq RC \leq$ core size $0 \leq RI \leq 0777$ $0 \leq RL \leq 03777$	No loading attempted.
ILL F/A DIR PARAM	Invalid parameter on assembler or FORTRAN control directive	Directive ignored. If SI = TY00, waits for input of next directive. If SI ≠ TY00, reads but ignores SI until next /JOB or /ENDJOB.
NO/EOF, NO LOAD ATTMPTD	No /EOF after an assembly or compilation, therefore, no end of file on GO file.	No loading attempted

SYSTEM LOADER

During loading, the events listed abort the loading procedure. In this case, if SI = TY00, MOS waits for the next directive. If SI ≠ TY00, SI is read but ignored until the next /JOB or /ENDJOB. When a map is requested, it is output prior to the termination message. For **size** and **missing program** errors, the names of all programs causing the error are also listed.

Message	Cause
LDR READ ERR	The loader encountered a reading error while attempting input of the object tape.
LDR RECORD ERR	The loader input an invalid type of record.
LDR CKSM ERR	The loader input an object text record with an invalid check-sum.
LDR SEQ ERR	The loader input an object text record with an invalid sequence number.
LDR TEXT ERR	The program object text contains an illegal or erroneous loader code.
LDR DATA ERR	The program attempted to overlay the loader, loader tables, or resident program; or the assembler attempted to overlay its I/O drivers.
LDR SIZE ERR	Program memory requirements exceed available program/common storage.
LIT POOL OVRFLW ERR	Program literal requirements exceed available literal storage.
LDR COMMON ERR	The programs contain conflicting size definitions for a common block.
MISSG PGRMS ERR	Loading requested named programs not found on either the binary or system file input devices.
LDR INITZTN ERR	Errors during loading of system loader.
NO EXCTN ADDR	Address to begin execution of the loaded program is missing from the object program.

I/O CONTROL

During program execution, calls to I/O control for input/output functions containing one of the following errors cause posting of a run-time error. In this case, if SI = TY00, MOS waits for the next directive. If SI ≠ TY00, SI is read but ignored until the next /JOB or /ENDJOB.

Message	Cause
ILL LUN/NAME USED IN DIR	An undefined logical unit specified as a parameter in an I/O control directive.
IOCS CALL TO UNASSGND LUNxx	An attempt to call an I/O driver not in memory (i.e., an unassigned logical unit specified by xx).

LANGUAGE PROCESSORS

During assembly and compilation, the following errors associated with language processors cause posting of an error message. In this case, if SI = TY00, MOS waits for the next directive. If SI ≠ TY00, SI is read but ignored until the next /JOB or /ENDJOB.

Message	Cause
PROCSSR ERR JOB ABORTD	Language violation.
PROCSSR RCRD COUNT ERR	The number of source statements read during pass 2 is not equal to that of pass 1.
LUN.xx SYS CNTRL DIR INPT ERR	A control directive (i.e., a slash in column 1) was read from unit xx prior to an END statement.
LUN.xx EOF ERR	An end of file was read from unit xx prior to an END statement.
LUN.xx END/BEG OF	The physical end or beginning device was encountered on unit xx prior to an END statement.
LUN.xx I/O ERR	An unrecoverable I/O error occurred on unit xx during assembly or compilation.

DAS MR ASSEMBLER

During assembly, the source statements are checked for syntax errors and usage. In addition, errors can occur where the program cannot determine the correct meaning of the source statement.

When an error is detected, the assembler outputs an error code following the source statement containing the error, on the LO, and continues to the next statement.

The assembler error messages (codes) are:

Code	Definition
*AD	An address expression is in error.
*DC	A decimal character appears in an octal constant.
*DD	There is an invalid redefinition of a symbol or the location counter.
*E	The symbolic source statement is incorrectly formed.
*EX	An expression contains an illegal construction.
*FA	A floating point number contains a format error.
*IL	The first non-blank character of the source statement is invalid; the statement is not processed.
*NR	No memory space available for the addition of an entry to the assembler's tables.
*NS	No symbol in the label field of a SET, EQU, MAC or FORM directive statement. No symbol in the label or variable field of an OPSY directive statement. No symbol in the variable field of a NAME directive statement.

*OP	The instruction field is undefined; two No Operation (NOP) instructions are generated in the object program. The remainder of the statement is not processed. Illegal nesting of DUP or MAC directive statements also causes this error.
*QQ	Illegal use of prime (').
*R	A relocatable item was encountered in the place where an absolute item was expected.
*SE	The symbol in the location field has a value during pass 2 that is different than the value used in pass 1.
*SY	An expression contains an undefined symbol.
*SZ	The expression value is too large for the size of the subfield or a DUP statement specifies that more than three symbolic source statements are to be assembled.
*TF	Undefined or illegal index register specification.
*UC	An undefined character in an arithmetic expression.
*UD	Undefined symbol in variable field of USE directive.
*XR	Address out of range for indexing specification.
* =	Illegal use of a literal.

FORTRAN IV COMPILER AND RUNTIME

COMPILER

During compilation, source statements are checked for such items as validity, syntax, and usage. When an error is detected, it is posted on the LO beneath the source statement. The errors marked T terminate binary output.

All error messages are of the form

ERR xx c(1)-c(16)

where xx is a number from 0 to 18 (notification error), or T followed by a number from 0 to 9 (terminating error); and c(1)-c(16) is the last character string (up to 16) encountered in the statement being processed. The right-most character indicates the point of error and the @ indicates the end of the statement. The possible error messages are:

Notification Error	Definition
0	Illegal character input
1	Construction error
2	Usage error
3	Mode error
4	Illegal DO termination
5	Improper statement number
6	Common base lowered
7	Illegal equivalence group
8	Reference to nonexecutable statement
9	No path to this statement
10	Multiply defined statement number
11	Invalid format construction
12	Spelling error
13	Format statement with no statement number
14	Function not used as variable
15	Truncated value
16	Statement out of order
17	More than 29 named common regions
18	Noncommon data

Terminating Error	Definition
T1	Construction error
T2	Usage error
T3	Data pool overflow
T4	Illegal statement
T5	Improper use
T6	Improper statement number
T7	Mode error
T8	Constant too large
T9	Improper DO nesting

RUNTIME

When an error is detected during runtime, a message is posted on the SO device and the job is aborted. The messages and their definitions are:

Message	Cause
ARITH OVFL	Arithmetic overflow
GO TO RANGE	Computer GO TO out of range
FUNC ARG	Invalid function argument (e.g., square root of negative number)
FORMAT	Error in FORMAT statement
MODE	Mode error (e.g., outputting real array with I format)
DATA	Invalid input data (e.g., inputting a real number from external medium with I format)
I/O	I/O error (e.g., parity, EOF)

FILE EDITING PROGRAM

During the file editing program, the following errors can occur:

Message	Cause	Action
INPUT FMT ERR n	An unrecognizable record read from SI (n = 1) or from the source file, PI, (n = 2)	No action taken on PO.
CATL OFLO	More than 20 entries in the catalog	The PO backspaces to the previous file, two EOFs are written and the job terminated.
OUTPUT EOM	The end of media detected on PO	Same as above.
END INPUT-FILE filename NOT FOUND	The second filename specified in a COPY control record not on the old source library (PI)	Two EOFs are written on PO and the job terminated.
FILE filename NOT FOUND	The filename is an EDIT control record, or the first filename in a COPY control record is not on PI.	The control directive is ignored and the job continues.
CTRL REC ERROR n	A control record is misplaced (n = 1), a required parameter is missing (n = 2), contains too many parameters (n = 3), or contains a parameter format error (n = 4)	If n = 1, action is described under INPUT FMT ERR n. If n = 2. 3. pr 4, the control record is ignored and the program continues.
END INPUT	An EOF or MOS control	Same as for INPUT FMT

Message	Cause	Action
	directive (/card) is read when a file editing control directive was expected.	ERR n.
SEQ. OFLO	While resequencing a file, more than 9,999 data records are read.	Sequencing restarts at zero and the program continues. NOTE: The records in this file can no longer be accessed by internal sequence numbers.
aaannnn NOT FOUND	While searching a file by internal sequence number, a sequence number higher than the one being searched for is encountered.	If the desired sequence number is the first number of a DEL or RPL group, or or is the sequence number specified in an ADD control record, the higher number is used in its place. If the desired sequence number is the second number of a DEL or RPL group, the record preceding the record containing the higher number is taken to be the last record of the group.
SEQ. ERROR aaannnn-aaannnn	While searching a file by internal sequence number, an out-of-sequence condition is found.	Same as for INPUT FMT ERR n.
CTRL. SEQ. ERROR	An internal sequence number or external line number lower than the previous one is in an ADD, DEL, or CHG control record.	Same as for INPUT FMT ERR n.

SYSTEM MAINTENANCE PROGRAM

During the system maintenance operation, the following errors can occur:

Message	Cause
CHECKSUM ERROR name	There is a checksum error in a record of the named program.
READ ERROR name	An error indication is received from I/O control after reading a record of the named program.
WRITE ERROR name	An error indication is received from I/O control after writing a record of the named program.
NO NAME name	The named program contains no entry name in the binary object module.
RECORD SIZE name	The size of a binary record as determined by I/O control for the named program is not 60 words (53 for the 622).
LOADER CODE ERROR name	The binary loader text for the named program contains a code or subcode not recognized by the loader.
SEQUENCE ERROR name	A sequence number in the binary object module for the named program is incorrect.
STRUCTURE ERROR name	There is a nonbinary record in the named object module.

After an error message is logged, enter one of the following statements:

Statement	Definition
RCRD	Reread the last record. If the error occurred on a magnetic tape, drum, or disc unit, the system maintenance program backspaces the record. Otherwise, manually position the record so it can be reread.
PGRM	Restart the program. If the error occurred on a magnetic tape, drum, or disc unit, the program backspaces to the beginning of the program. Otherwise, manually position the program so it can be reread.
SMAIN	Restart the system maintenance program.

Key-in errors result in a repeat of the error message:

The system then waits for a correct entry.

SYSTEM PREPARATION

During system preparation, the following errors can occur. The first two can be corrected by reentering the information noted.

Message	Cause	Action
ILLEGAL PREP DIRECTIVE	There is a syntax error in the control directive just read.	Reenter the directive through the TTY.
DEVICE NAME NOT VALID ENTER DEVICE NAME FOR xx	There is a syntax error in a peripheral device name.	Reenter the name through the TTY.

The errors listed below are corrected by the procedure given at the end of the list.

Message	Cause
SEQUENCE ERROR name	A sequence number in the object module for the named program is incorrect.
STRUCTURE ERROR name	There is a nonbinary record in the named object module.
LITERAL POOL OVERFLOW name	The size allocated to the literal pools is exceeded during the generation of the named absolute module.
COMMON ERROR name	The size of a common block is illegally redefined during the generation of the named absolute program.
PROGRAM SIZE ERROR name	The size of the named absolute program exceeds the available memory.
MEMORY SIZE ERROR name	The tables internal to the system preparation program during the generation of the absolute program named exceed the available memory.
MISSING PROGRAMS ERROR name	During the generation of the absolute program named, there is an external reference that cannot be satisfied before encountering ENDABS.

NO EXECUTE ADDRESS name	No program execution address is found for any program during the generation of the absolute program named.
CHECKSUM ERROR name	There is a check-sum error in a record of the named program.
READ ERROR name	An error, end, or beginning of device is received from I/O control after a reading operation for the named program.
WRITE ERROR name	An error, end or beginning of device, or end of file is received from I/O control after a writing operation for the named program.
NO NAME name	The named program contains no name in the object module.
LOADER CODE ERROR name	The object module for the named program contains a code or subcode not recognized by the loader.

After an error message is logged, enter one of the following statements:

Statement	Definition
RCRD	Reread the last record. If the error occurred on a magnetic tape, drum, or disc unit, the system preparation program backspaces the record. Otherwise, manually position the record so it can be reread. This procedure does not apply to errors during ABS processing.
PGRM	Restart at the beginning of the current binary loader text program. If the error occurred on a magnetic tape, drum or disc unit, the system preparation program backspaces to the beginning of the program. Otherwise, manually position the program so it can be reread.

Key-in errors result in a repeat of the error message:

The system then waits for a correct entry.

SECTION 14 - MOS FORMATS

ABSOLUTE MODULE FORMAT

Absolute modules are created by the system preparation program to construct the system file. The modules are made up of text and string records.

Word 1 of each record is a control word, as follows:

Bit	Binary Value	Definition
16-17	--	622-series computers only: unused
15	0	Verify the record check-sum
	1	Suppress check-sum verification
13-14	11	Binary record
	00-10	Nonbinary record
12	0	First record of a module
	1	Not the first record of a module
11	0	Last record of a module -- execution address in the word following the last word of text
	1	Not the last record of a module
10	0	Text record
	1	String record
9	--	Unused
8	0	Absolute module
	1	Not an absolute module (e.g., relocatable object module)
0-7	Any	Sequence number

Word 2 contains the exclusive-OR check-sum of words 1 and 3 through 60.

Word 3 contains zeros when the first record of a module. When not the first record, it contains text.

Words 4 through 7 in the first record of a module contain the module (program) name as left-justified, blank-filled ASCII characters. When not in the first record, these words contain text.

Words 8 through 11 in the first record contain the creation date of the module (program) in ASCII characters. When not in the first record, these words contain text.

Words 12 through 60 contain text.

Text data on text records (word 1, bit 10 = 0) is organized into a variable number of variable length blocks as follows:

Word	Contents
3	Number of words of text that follow (n-1) in block 1
4	Starting load location for the text that follows in block 1
5	Text word 1 in block 1
.	.
.	.
.	.
4+n	Text word n in block 1
.	.
5+n	Number of words of text that follow (n-1) in block 2
6-n	Starting load location for text that follows in block 2
7-n	Text word 1 in block 2
.	.
.	.
.	.
6+n+n'	Text word n in block 2
.	.
.	.
.	.

A text record can contain a number of text words equal to or less than the capacity of one card, i.e., up to 57 words. However, it will never contain a partial block. When a record is not full, the last *number of words of the text minus one* value is negative to indicate the logical end of the record.

String data on string records (word 1, bit 10 = 1) is organized into a variable number of variable length blocks of information, as follows:

Word	Contents
3	Number of words of string entry points minus one (n-1) on this record.
4	String entry point 1
5	String reference 1, 1
.	.
.	.
.	.
4-m	One's complement of string reference 1,m
.	.
.	.
.	.
5-m	String entry point 2
6-m	String reference 2,1
.	.
.	.
.	.
6-m-m'	One's complement of string reference 2,m

OBJECT MODULE FORMAT

Object modules generated by the MOS language processors result from assembly or compilation. The modules are input by the MOS loader and are bound together into an executable program.

The first record of the module contains the size of the program, an eight-character identification, and an eight-character date. Entry name addresses, if any, appear as the first data field items of the object module.

RECORD STRUCTURE

Sixteen-Bit Computers

Object module records have a fixed length of sixty 16-bit words. Word 1 is the record control word. Word 2 contains the exclusive-OR check-sum of word 1 and words 3 to 60. Words 3 to 11 can contain a program identification block (optional). Words 12 to 60 (or 3 to 60 if there is no program identification block) contain data fields.

Eighteen-Bit Computers

Object module records have a fixed length of fifty-three 18-bit words. Word 1 is the record control word. Word 2 contains the exclusive-OR check-sum of word 1 and words 3 through 53. Words 3 through 11 can contain a program identification block (optional). Words 12 through 53 (or 3 through 53 if there is no program identification block) contain data fields.

Table 14-1 illustrates record-control word formats.

PROGRAM IDENTIFICATION BLOCK

The program identification (ID) block appears in words 3 to 11 of the starting record of each module. Word 3 contains the program size, words 4 to 7 contain an ASCII eight-character program identification, and words 8 to 11 contain an ASCII eight-character date. The program ID is the job name at the time that the object module is assembled.

DATA FIELD FORMATS

Data fields contain one-, two-, three-, or four-word entries. One-word entries consist of a control word; two-word entries consist of a control word and a data word; three-word entries consist of a control word and two data words; and four-word entries consist of a control word, two name words, and a data word. Data words can contain instructions, constants, chain addresses, entry addresses, and address offset values.

Table 14-1. Record Control Word Format

Bit	Binary Value	Meaning
16-17	00	Undefined (18-bit computers only)
15	0	Verify check-sum
	1	Suppress check-sum
13-14	11	Binary record
	00-10	Nonbinary record
12	0	First record of module
	1	Not the first record
11	0	Last record of module
	1	Not the last record
10	0	
9	0	
8	0	Not a relocatable module (i.e., absolute)
	1	Relocatable module
0-7		Sequence number (modulo 255)

LOADER CODES

Loader codes, which have the following format, are among the data in an object module.

```
17 16 15    14 13 12    11 10 9    8 7 6    5 4 3   2 1 0
          ----------  ----------  ----------  ---------
622/i   Code        Subcode       Pointer     Name
```

Code Values	Meaning
00	Refer to subcode for specific action.
01	Undefined.
02	Add the value of the selected pointer to the data word before loading.
03	Add the value of the selected pointer to the first data word (literal value) and enter the sum in the direct literal pool if bit 11 of the second data word is zero. Otherwise, enter it in the indirect literal pool. Add the address of the literal to the second data word before loading.
04	Load the data word(s) absolute. Bits 12 through 0 indicate the number of words minus one (n-1) to load.
05-07	Undefined.

Subcode Values	Meaning
00	Ignore this entry (one word only).
01	Set the loading address counter to the sum of the specified pointer plus the data word.
02	Chain the current loading address counter value to the chain whose last address is given by the sum of the selected pointer plus the data word. Stop chaining when an absolute zero address is encountered.
03	Complete the postprogram references by adding to each address the sum of the selected pointer plus the data word.
04	Undefined.

05	Load the I/O driver currently associated with the logical device number specified by the sum of the selected pointer plus the data word.
06	Undefined.
07	Set the program execution address to the sum of the values of the selected pointer plus the data word.
010	Define the entry name with entry location as equal to the value of the selected pointer plus the data word.
011	Define a region for the pointer whose size is given in the data word. If the entry name is not blank, define the entry point as the base of the region.
012	Enter a load request for the external name. The chain address is given by the sum of the selected pointer plus the data word.
013	Enter the loading address of the external name in the indirect literal pool. Add the address of the literal plus the value of the selected pointer to the data word (command) before loading.
014-017	Undefined.

Pointer Values	Meaning
00	Program region.
01	Postprogram region.
02	Blank common region.
03-036	Labelled COMMON regions.
037	Absolute (no relocation).

Name Format

Names are one to six (six-bit) characters, starting in bit 3 of the control word and ending with bit 0 of the second name word. Only the right 16 bits of the two name words are used.

EXAMPLE

The following is a sample program with description of the object module format after assembly and the core image after loading.

Source Module

```
                NAME        SUBR
                EXT         BBEN
SUBR            ENTR
                LDA*        SUBR
                CALL        BBEN
                STA         TIME
                JAN         DONG
                LDA         =2
                CALL        BBEN
DONG            INR         SUBR
                JMP*        SUBR
TIME            BSS         I
                END
```

Object Module

060400	Record control word (first and last record, verify check-sum sequence number 0)
157631	Check-sum word.
	(Begin program tD block)
000016	Program size (exclusive of FORTRAN COMMON, literals, and indirect address pointers).
142730 140715 150314 142640	Identification in ASCII (assume this program was labeled EXAMPLE).
131263 126661 130255 133271	Date of creation in ASCII (assume assembled 03-10-69)
	(End program ID block)
010000 000647 054262 000000	Define entry name SUBR at relative 0 (code 0, subcode 010, pointer 0, name SUBR, and data word 0).
100000 000000	Enter absolute data word 0 in memory at relative 0.

060000 100000 017000	Enter literal (indirectly addressed relative 0) in indirect pointer pool, add address of pointer to load 017000 and enter memory at relative 1.
100000 002000	Enter absolute data word 02000 in memory at relative 2.
100000 000000	Enter absolute data word 000000 in memory at relative 3.
100000 054010	Enter absolute data word 054010 in memory at relative 4.
100000 001004	Enter absolute data word 01004 in memory at relative 5.
040000 000012	Enter relative data word 012 in memory at relative 6.
060760 000002 010000	Enter literal (absolute 2) into literal pool, add address of literal to load command 010000, and enter in memory at relative 7.
100000 002000	Enter absolute data word 02000 in memory at relative 010.
040000 000003	Enter relative data word 03 in memory at relative 011.
060000 000000 047000	Enter literal (relative 0) into indirect pointer pool, add address of literal to increment command 047000, and enter in memory at relative 012.
100000 001000	Enter absolute data word 01000 in memory at relative 013.
040000 100000	Enter relative data word 0100000 in memory at relative 014.
001000	Set loading location for next command, if any, to relative 016.
012003 000212 024556	Enter load request for external name BBEN and chain entry address to relative 011. 000011

(The remaining words of this record contain zero.)

Core Image

Assume the program originates at 0500, the literal pool limits are 0200-0400, and BBEN is loaded at 0516.

```
0200        100500      DATA        0500
0201        000500      DATA        0500
    .
    .
    .
0377        000002      DATA        2
    .
    .
    .
0500        000000      ENTR        0
0501        017200      LDA*        0200
0502        002000      JMPM
0503        000516                  516
0504        054010      STA         0515
0505        001004      JAN
0506        000512                  0512
0507        010377      LDA         0377
0510        002000      JMPM
0511        000516                  0516
0512        047201      INR*        0201
0513        001000      JMP
0514        100500          *       0500
0515                    BSS         1
0516                    BSS         1
```

The following six-bit codes are used by the loader in building object modules. The codes define names created by NAME, EXT, and /JOB directives.

Character	Octal	Character	Octal	Character	Octal
@	40	V	66	+	13
A	41	W	67	,	14
B	42	X	70	-	15
C	43	Y	71	.	16
D	44	Z	72	/	17
E	45	[	73	0	20
F	46	\	74	1	21
G	47	]	75	2	22
H	50	↑	76	3	23
I	51	←	77	4	24
J	52	(blank)	00	5	25
K	53		01	6	26
L	54	–	02	7	27
M	55		03	8	30
N	56	$	04	9	31
O	57		05	:	32
P	60	&	06	;	33
Q	61	'	07	<	34
R	62	(	10	=	35
S	63	)	11	>	36
T	64	*	12	?	37
U	65				

DATA FORMAT

This appendix explains the formats and symbols used by MOS for storing information on cards and paper tape.

PAPER TAPE

Information stored on paper tape is either binary or alphanumeric. It is separated into records (blocks of words) by three blank frames. The last frame of each record contains an end-of-record mark (1-3-4-8 punch).

Binary Mode

Binary information is stored with three frames per computer word (figure 14-2). Note that channels 6 and 7 are always punched.

Alphanumeric Mode

Alphanumeric and unformatted binary information is stored with one frame per character or one frame per 8 bits of unformatted binary data (figure 14-3). Standard ASCII-8 punch levels are used.

Special Characters

An end of file is represented by the ASCII-8 BELL character (1-2-3-8 punch).

When paper tape is punched on a teletypewriter, the ASCII-8 ERROR character flags erroneous frames punched by the teletypewriter when it is turned on or off. This notifies the TTY and paper tape reader drivers to ignore the next frame.

When alphanumeric input tapes are punched off-line on a teletypewriter, there is no means of spacing the three blank frames after every record. The following procedure gives a tape that can be read by the TTY reader and paper tape reader drivers:

a. Punch the alphanumeric statement.

b. Punch an end of record (RETURN on the TTY keyboard).

c. Punch three or more frames of the line-feed character.

d. Punch the next alphanumeric statement. Return to step b.

CARDS

Information stored on cards is either binary or alphanumeric. Each card holds one record of information. Hence, there is no end-of-record character for cards.

Binary Mode

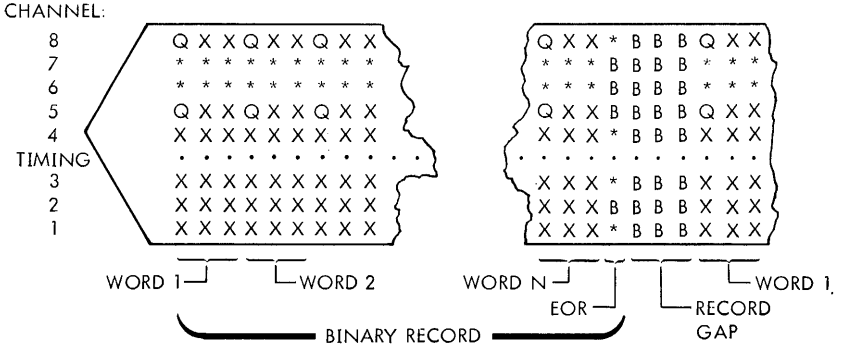
Binary information is stored with sixty 16-bit words or fifty-three 18-bit words per card. The information is serial with bit 15 of the first word in row 12 of column 1, bit 14 in row 11, etc. (figure 14-4). Note that 18-bit records occupy only the first 954 bits on the card (i.e., the last six bits in column 80 are not used).

Alphanumeric Mode

Alphanumeric information is stored one character per card column (figure 14-5) using the standard punch patterns.

Special Character

An end of file is represented on cards by a 2-7-8-9 punch in column 1 of an otherwise blank card.



* = HOLE
B = BLANK
X = DATA BIT
EOR = END - OF - RECORD
Q = BLANK FOR 16-BIT WORD; DATA BITS #17 AND #16 FOR 18-BIT WORD.

Figure 14-2. Paper Tape Binary Record Format

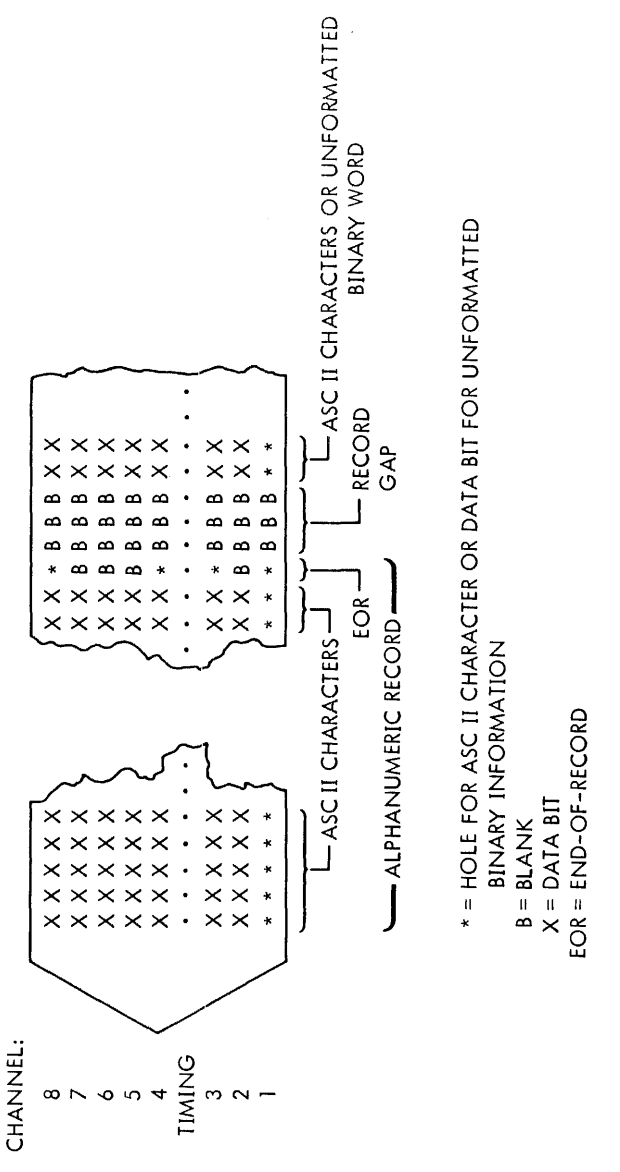
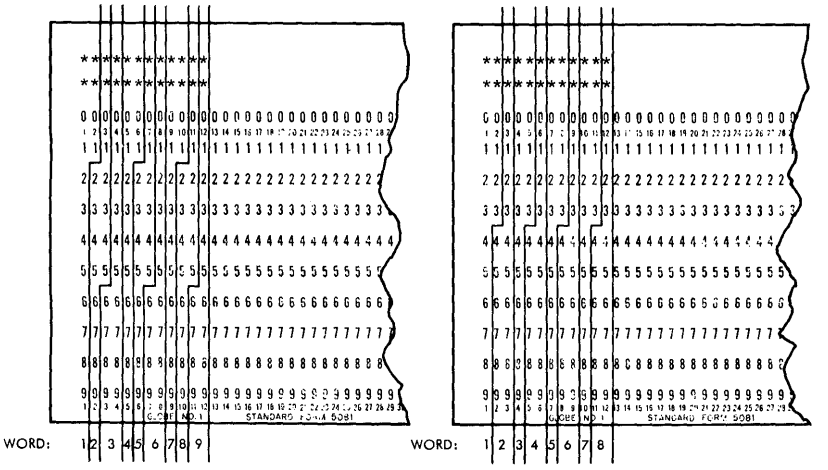


Figure 14-3. Paper Tape Alphanumeric Record Format



A. 16-BIT WORD FORMAT

B. 18-BIT WORD FORMAT

Figure 14-4. Card Binary Record Format

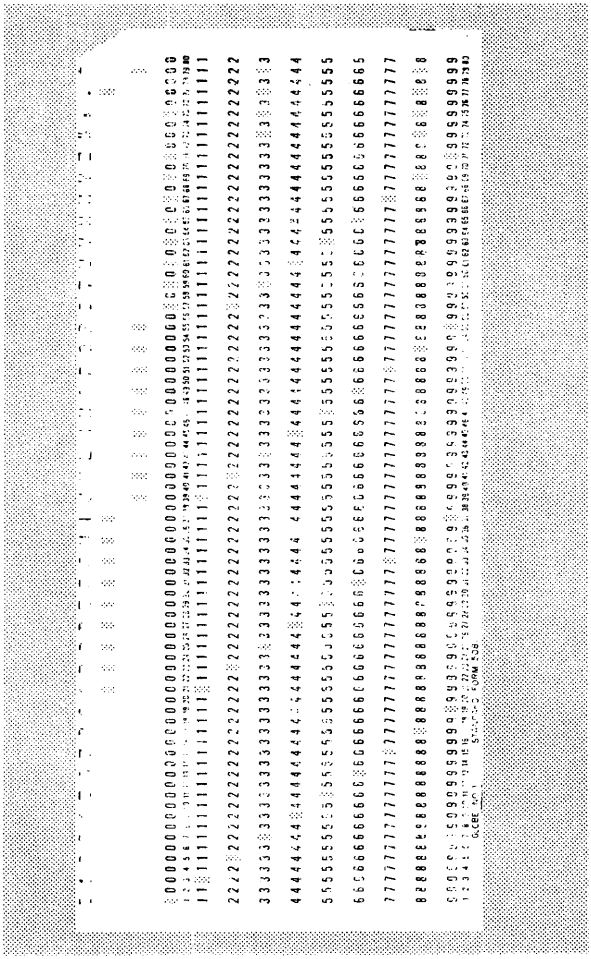


Figure 14-5. Card Alphanumeric Record Format (IBM 026)

APPENDIX TTY CHARACTER CODES

Character	Internal Code	Character	Internal Code
0	260	Y	331
1	261	Z	332
2	262	(blank)	240
3	263		241
4	264	'	242
5	265	#	243
6	266	$	244
7	267		245
8	270	&	246
9	271	'	247
A	301	(	250
B	302	)	251
C	303	*	252
D	304	+	253
E	305	,	254
F	206	—	255
G	307	.	256
H	310	/	257
I	311	:	272
J	312	;	273
K	313		274
L	314	=	275
M	315		276
N	316		277
O	317	@	300
P	320		333
Q	321		334
R	322		335
S	323		336
T	324		337
U	325	RUBOUT	377
V	326	NUL	200
W	327	SOM	201
X	330	EOA	202

Character	Internal Code	Character	Internal Code
EOM	203	X-OFF	223
EOT	204	TAPE OFF	
WRU	205	AUX	224
RU	206	ERROR	225
BEL	207	SYNC	226
FE	210	LEM	227
H TAB	211	S0	230
LINE FEED	212	S1	231
V TAB	213	S2	232
FORM	214	S3	233
RETURN	215	S4	234
SO	216	S5	235
SI	217	S6	236
DCO	220	S7	237
X-ON	221		
TAPE AUX ON	222		