SC28-6478-3

Program Product

IBM DOS/VS COBOL Compiler and Library Programmer's Guide

Program Numbers: 5746-CB1 (Compiler and Library)
5746-LM4 (Library)

IBM

Fourth Edition (February 1979)

This is a major revision of, and obsoletes, SC28-6478-0, -1, and -2. It also obsoletes their technical newsletters SN28-1063, SN20-9121, SN20-9141, SN20-9180, and SN20-9235.

This edition corresponds to Release 2 of the IBM DOS/VS COBOL Compiler and Library, program numbers 5746-CB1 and 5746-LM4.

Changes are continually made to the information herein; before using this publication in connection with the operation of IBM systems, consult the latest *IBM System/370 Bibliography,* GC20-0001, for the editions that are applicable and current.

Publications are not stocked at the address given below; requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

Comments may be addressed to IBM Corporation, P.O. Box 50020, Programming Publishing, San Jose, California U.S.A. 95150. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

<u>PREFACE</u>

This publication describes how to compile a COBOL program using the Program Product IBM DOS/VS COBOL Compiler. It also describes how to link edit the resulting object module, and execute the program. Included is a description of the output from each of these three steps: compile, link edit, and execute. This publication explains features of the DOS/VS Compiler and Library, and available options of the operating system.

This publication is logically and functionally divided into four parts. Part I contains information useful to programmers who are running COBOL programs compiled on the DOS/VS Compiler, under the control of the IBM Disk Operating System Virtual Storage. Part I covers such topics as job control language, library usage, and interpreting output.

Part II contains supplemental information on the use of the language as specified in the publication <u>IBM DOS Full American National Standard COBOL</u>, GC28-6394, and should be used in conjunction with this publication for coding COBOL programs. Part II covers in detail such topics as file organization, file label handling, and record formats. Part II is intended as reference material for language features that are primarily system-dependent.

Part III contains information on programming techniques useful to the programmer running COBOL programs compiled on the DOS/VS Compiler. Topics such as coding considerations, table handling considerations, and formatting data are covered in Part III.

Part IV contains error determination information. This part covers such topics as program debugging and program testing.

Diagnostic messages generated by the DOS/VS Compiler and Library and their accompanying documentation can be found in this publication.

Information on installing the DOS/VS Compiler and Library can be found in the following publication:

<u>IBM DOS/VS COBOL Compiler and Library, Installation Reference Material</u>, SC28-6479

Wider ranging and more detailed discussions of the DOS/VS System are given in the following publications:

<u>Introduction to DOS/VS</u>, GC33-5370

<u>DOS/VS System Generation</u>, GC33-5377

<u>DOS/VS System Management Guide</u>, GC33-5371

<u>DOS/VS Data-Management-Concepts, GC24-5138</u>

<u>DOS/VS Supervisor and I/O Macro Reference</u>, GC33-5373

<u>DOS/VS System Control Statements</u>, GC33-5376

<u>DOS/VS Access Method Services</u>, GC33-5382

<u>DOS/VS System Utilities Reference</u>, GC33-5381

<u>DOS/VS Messages</u>, GC33-5379

The following publications provide detailed information on the IBM 3886 Optical Character Reader:

<u>IBM 3886 Optical Character Reader General Information Manual</u>, GA21-9146

<u>IBM 3886 Optical Character Reader Input Document Design and Specifications</u>, GA21-9148

<u>DOS/VS Planning Guide for the IBM 3886 Optical Character Reader, Model 1</u>, GC21-5059

The following publications provide information on the IBM DOS/VS Sort/Merge Program Product, Program Number 5746-SM1, and the DOS Sort/Merge Program Product, Program Number 5743-SM1:

<u>IBM DOS/VS Sort/Merge General Information</u>, GC33-4030

<u>IBM DOS/VS Sort/Merge Program Product Design Objectives</u>, GC33-4027

<u>IBM DOS/VS Sort/Merge Installation Reference Material</u>, SC33-4026

<u>IBM DOS Sort/Merge Programmer's Guide</u>, SC33-4018

The titles and abstracts of related publications are listed in <u>IBM System/360 and System/370 Bibliography</u>, GA22-6822.

Summary of Amendments Number 5

Form of Publication: Revision SC28-6478-3

New: Programming Function

Support for fixed block devices is provided under DOS/VSE with VSE/Advanced Function, Release 1.

Maintenance: Documentation

Clarifications and corrections have been made in various areas of the text.

Editorial changes that have no technical significance are not noted here.

Specific changes to the text made as of this publishing date are indicated by a vertical bar to the left of the text. These bars will be deleted at any subsequent republication of the page affected.

Summary of Amendments — Number 4

Form of Publication: TNL SN20-9235 to SC28-6478-0, -1, -2

New: Programming Function

Support has been added for the 3330-11 Disk Storage and 3350 Direct Access Storage devices.

Maintenance: Documentation

Minor technical changes and additions have been made to the text.

Summary of Amendments — Number 3

Date of Publication: December 3, 1976

Form of Publication: TNL SN20-9180 to SC28-6478-0, -1, -2

IBM DOS/VS COBOL

Maintenance: Documentation

Minor technical changes and additions have been made to the text.

Editorial changes that have no technical significance are not noted here.

Specific changes to the text made as of this publishing date are indicated by a vertical bar to the left of the text. These bars will be deleted at any subsequent republication of the page affected.

Summary of Amendments — Number 2

Date of Publication: January 9, 1976

Form of Publication: SN20-9141 to SC28-6478-0, -1

Support has been added to run DOS/VS COBOL under control of VM/370 CMS Release 3.

DOS/VS COBOL programs can be compiled in CMS and then executed in a DOS virtual machine, or under a DOS system.

The following restrictions apply to execution of DOS/VS COBOL programs in CMS:

1. Indexed files (DTFIS) are not supported. Various clauses and statements are therefore invalid: RECORD KEY, APPLY CYL-OVERFLOW, NOMINAL KEY, APPLY MASTER/CYL-INDEX, TRACK-AREA, APPLY CORE-INDEX, and START.
2. Creating direct files is restricted as follows:
   –For U or V recording modes, access mode must be sequential.
   –For ACCESS IS SEQUENTIAL, track identifier must not be modified.
3. None of the user label-handling functions are supported. Therefore, the label-handling format of USE is invalid. The data-name option of the LABEL RECORDS clause is invalid.
4. There is no Sort or Segmentation feature.
5. ASCII-encoded tape files are not supported.
6. Spanned records (S-mode) processing is not available. This means that the S-mode default (block size smaller than record size) cannot be specified, and that the RECORDING MODE IS S clause cannot be specified.

In addition, multitasking, multipartition operation, and teleprocessing functions are not supported when executing under CMS.

For a more detailed description of VM/370 CMS for DOS/VS COBOL, see *IBM VM/370 CMS User's Guide for COBOL,* order number SC28-6469.

Summary of Amendments — Number 1

Date of Publication: March 22, 1974

Form of Publication: TNL SN28-1063 to SC28-6478-0

New: Additional Compiler Capabilities

Lister feature
Execution Statistics and
Verb summary feature

SORT-OPTION

Maintenance: Documentation Only

Minor technical changes and corrections.

Editorial changes that have no technical significance are not noted here.

Specific changes to the text made as of this publishing date are indicated by a vertical bar to the left of the text. These bars will be deleted at any subsequent republication of the page affected.

ILLUSTRATIONS

TABLES

The IBM DOS/VS COBOL Compiler includes the following features:

- Object Code:

  (1) Optimized Object Code -- which results, when specified, in up to 30% space saving in object program generated code and global tables as compared with Version 2 of the IBM DOS Full American National Standard COBOL Compiler. The space saved depends on the number of referenced procedure-names and branches, and on 01-level data names.

  (2) Double-Buffered ISAM -- allows faster sequential processing of indexed files.

  (3) The MOVE Statement and Comparisons -- when a MOVE statement or a comparison involves a one-byte literal, generated code for the move and the comparison saves object program space and compilation time.

  (4) DISPLAY Routines -- the DISPLAY routine has been split into subsets for efficient object program code.

- Alphabetized Cross-Reference Listing (SXREF) -- for easier reference to user-specified names in a program. SXREF performs up to 25 times faster than the source-ordered cross-reference (XREF) feature of Version 2 of the IBM DOS Full American National Standard COBOL Compiler. The larger the source program, the more that performance is improved. Total compilation time is up to 2 times faster.

- Debugging Facilities:

  (1) Symbolic Debug Feature -- which provides a symbolic formatted dump at abnormal termination, or a dynamic dump during program execution.

  (2) Flow Trace Option -- a formatted trace can be requested for a variable number of procedures executed before abnormal termination.

  (3) Statement Number Option -- identifies the COBOL statement being executed at abnormal termination.

  (4) Expanded CLIST and SYM -- for detailed information about the Data Division and Procedure Division.

  (5) Relocation Factor -- can be requested to be included in addresses on the object code listing, for easier debugging.

  (6) Working-Storage Location and Size -- When CLIST and SYM are in effect, the starting address and size of Working-Storage are printed.

  (7) Syntax-Check Feature -- optionally provides a quick scan of the source program without producing object code. Syntax checking can be conditional or unconditional.

  (8) WHEN-COMPILED Special Register -- makes the date-and-time-compiled constant carried in the object module available to the object program. This special register is a programmer aid that provides a means of associating a compilation listing with both the object program and the output produced at execution time.

- Device Support -- the following devices can be specified in addition to devices supported by the IBM DOS Full American National Standard COBOL compilers:

  5203,3203 -- line printers

  3211 -- 150-character printer

  3330,3340,3350 -- mass storage (direct access) facilities

  Fixed block direct access storage devices

  3540 -- Diskette input/output unit

  3410,3420 -- tape utility devices

  2560,3504,3505,3525,3881,3886,5425 -- advanced unit-record devices

- ASCII Support -- allows creation and retrieval of tape files written in the American National Standard Code for Information Interchange (ASCII).

- VSAM (Virtual Storage Access Method) Support -- provides fast storage and retrieval of records, password protection, centralized and simplified data and space management, advanced error recovery facilities, plus system catalog. COBOL supports indexed (key-sequenced) files and sequential (entry-sequenced) files. Records can be fixed or variable in length.

- FIPS (Federal Information Processing Standard) Flagger -- issues messages identifying nonstandard elements in a COBOL source program. The FIPS Flagger makes it possible to ensure that COBOL clauses and statements in a DOS/VS COBOL source program conform to the Federal Information Processing Standard.

- Lister -- provides a specially formatted source listing with embedded cross-references for increased intelligibility and ease of use. A reformatted source deck is available as an option.

- Generic Key Facility for ISAM Files -- sequential record retrieval can be requested using a search argument comprised of a user-specified number of high-order characters (generic portion) of the NOMINAL KEY. The user need not specify a full or exact search key. This feature is supported via the START verb.

- MERGE Support -- combines from two to eight identically sequenced files on a set of specified keys and makes records available, in merged order, to an output procedure or a sequential output file.

- Verb profiles -- facilitates identifying and locating verbs in the COBOL source program. Options provide a verb summary or a verb cross-reference listing which includes the verb summary.

- Execution-time statistics -- maintains a count of the number of times each verb in the COBOL source program is executed during an individual program execution.

PART I

COBOL has undergone considerable refinement and standardization since 1959. A standard COBOL has been approved by the American National Standards Institute, an industry-wide association of computer manufacturers and users. This standard is called American National Standard COBOL. IBM Full American National Standard COBOL is compatible with American National Standard COBOL and includes a number of extensions to it as well.

An IBM COBOL program may be processed by the IBM DOS/VS System. Under control of the operating system, a set of COBOL source statements is translated to form a _module_. In order to be executed, the module in turn must be processed to form a _phase_. The reasons for this will become clear later. For now it is sufficient to note that the flow of a COBOL program through the operating system is from source statements to module to phase.

The DOS/VS System consists essentially of a _control program_ and a number of _processing programs_, and _data management_.

CONTROL PROGRAM

The components of the control program are: the Supervisor, Job Control Processor, and the Initial Program Loader.

SUPERVISOR

The main function of the Supervisor is to provide an orderly and efficient flow of jobs through the operating system. (A job is some specified unit of work, such as the processing of a COBOL program.) The Supervisor loads into the computer the phases that are to be executed. During execution of the program, control usually alternates between the Supervisor and the processing program. The Supervisor, for example, handles all requests for input/output operations.

JOB CONTROL PROCESSOR

The primary function of the Job Control Processor is the processing of job control statements. Job control statements describe the jobs to be performed and specify the programmer's requirements for each job. Job control statements are written by the programmer using the job control language. The use of job control statements and the rules for specifying them are discussed later.

INITIAL PROGRAM LOADER

The Initial Program Loader (IPL) routine loads the Supervisor into storage when system operation is initiated. Detailed information about the Initial Program Loader need not concern the COBOL programmer. Anyone interested in this material, however, can find it in the publication _DOS/VS System Management Guide_.

PROCESSING PROGRAMS

The processing programs include the COBOL compiler, service programs, and application programs.

SYSTEM SERVICE PROGRAMS

The system service programs provide the functions of generating the system, creating and maintaining the library sections, and editing programs into disk residence before execution. The system service programs are:

1. _Linkage Editor_. The Linkage Editor processes modules and incorporates them into phases. A single module can be edited to form a single phase, or several modules can be edited or _linked_ together to form one executable phase. Moreover, a module to be processed by the Linkage Editor may be one that was just created (during the same job) or one that was created in a previous job and saved.

   The programmer instructs the Linkage Editor to perform these functions through job control statements. In addition, there are several linkage editor control statements. Information on their use is given later.

2. Librarian. The Librarian consists of a group of programs used for generating the system, maintaining and reorganizing the disk library areas, and providing printed and punched output from the libraries. The system libraries are: the core image library, the relocatable library, the source statement library, and the procedure library. In addition, the Librarian supports private core image, relocatable, and source statement libraries. Detailed information on the Librarian is given later.

APPLICATION PROGRAMS

Application programs are usually programs written in a higher-level programming language (e.g., COBOL). All application programs within the Disk Operating System/Virtual Storage are executed under the supervision of the control program.

IBM-SUPPLIED PROCESSING PROGRAMS

The following are examples of IBM-supplied processing programs:

1. Language translators, e.g., DOS/VS COBOL, which translate source programs written in various languages into machine (or object) language.
2. Sort/Merge
3. Utilities

DATA MANAGEMENT

A third important class of components is data management routines. These are available for inclusion in problem programs to relieve the programmer of the detailed programming associated with the transfer of data between programs and auxiliary storage.

MULTIPROGRAMMING

Multiprogramming refers to the ability of the system to control more than one program concurrently by interleaving their execution. This support is referred to as fixed partitioned multiprogramming, since the virtual address space is divided into a fixed number of partitions. Each program occupies a contiguous area of storage. The amount of virtual storage allocated to programs to be executed may be determined when the system is generated, or it may be determined by the operator when the program is loaded into storage for execution.

BACKGROUND VS. FOREGROUND PROGRAMS

There are two types of problem programs in multiprogramming: background and foreground. Background and foreground programs are initiated by the Job Control Processor from batched-job input streams.

Background and foreground programs initiate and terminate independently of one another. Neither is aware of the other's status or existence.

The system is capable of concurrently operating one background program and four foreground programs. Priority for CPU processing is controlled by the Supervisor with foreground programs normally having priority over background programs. Control is taken away from a high priority program when that program encounters a condition that prevents continuation of processing, until a specified event has occurred. Control is taken away from a lower priority program when an event for which a higher priority program was waiting has been completed. Interruptions are received and processed by the Supervisor.

In a multiprogramming environment, the DOS/VS COBOL compiler can execute either in the background or the foreground. In systems that support the batched-job foreground and private core image library options, the Linkage Editor can execute in any foreground partition as well as in the background partition. To execute the DOS/VS COBOL compiler for the linkage editor in any foreground partition, a private core-image library is required. Additional information on executing the compiler and Linkage Editor in the foreground is contained in "Appendix F: System and Size Considerations." COBOL program phases can be executed as either background or foreground programs.

A job is a specified unit of work to be performed under control of the operating system. A typical job might be the processing of a COBOL program -- compiling source statements, editing the module produced to form a phase, and then executing the phase. Job definition -- the process of specifying the work to be done during a single job -- allows the programmer considerable flexibility. A job can include as many or as few job steps as the programmer desires.

JOB STEPS

A job step is exactly what the name implies -- one step in the processing of a job. Thus, in the job mentioned above, one job step is the compilation of source statements; another is the link editing of a module; another is the execution of a phase. In contrast to a job definition, the definition of a job step is fixed. Each job step involves the execution of a program, whether it be a program that is part of the Disk Operating System/Virtual Storage or a program that is written by the programmer. A compilation requires the execution of the DOS/VS COBOL compiler. Similarly, an editing implies the execution of the Linkage Editor Finally, the execution of a phase is the execution of the problem program itself.

Compilation Job Steps

The compilation of a COBOL program may necessitate more than one job step (more than one execution of the DOS/VS COBOL compiler). In some cases, a COBOL program consists of a main program and one or more subprograms. To compile such a program, a separate job step must be specified for the main program and for each of the subprograms. Thus, the DOS/VS COBOL compiler is executed once for the main program and once for each subprogram. Each execution of the compiler produces a module. The separate modules can then be combined into one phase by a single job step -- the execution of the Linkage Editor.

For a COBOL program that consists of a main program and two subprograms, compilation and execution require five steps: (1) compile (main program), (2) compile (first subprogram), (3) compile (second subprogram), (4) link edit (three modules combined into one phase), and (5) execute (phase). Figure 1 shows a sample structure of the job deck for these five job steps. Compilation and execution in three job steps -- compile, link edit, and execute -- is applicable only when the COBOL source program is a single main program.

```
// JOB PROG1
.
.
.
// EXEC FCOBOL
   {source deck - main program}
/*
.
.
.
// EXEC FCOBOL
   {source deck - first subprogram}
/*
.
.
.
// EXEC FCOBOL
   {source deck - second subprogram}
/*
.
.
.
// EXEC LNKEDT
.
.
.
// EXEC
```

Figure 1. Sample Structure of Job Deck for Compiling, Link Editing, and Executing a Main Program and Two Subprograms

Multiphase Program Execution

The execution of a COBOL program has thus far been referred to as the execution of a phase. It is possible, however, to organize a COBOL program so that it is executed as two or more phases. Such a program is known as a multiphase program.

By definition, a phase is that portion of a program that is loaded into virtual storage by a single operation of the Supervisor. A COBOL program can be executed as a single phase only if there is an area of virtual storage available to accommodate all of it. A program that is too large to be executed as a single phase must be structured as a multiphase program. The technique that enables the programmer to use subprograms that do not fit into virtual storage (along with the main program) is called _overlay_.

The number of phases in a COBOL program has no effect on the number of job steps required to process that program. As will be seen, the Linkage Editor can produce one or more phases in a single job step. Similarly, both single-phase and multiphase programs require only one execution job step. Phase execution is the execution of all phases that constitute one COBOL program.

Detailed information on overlay structures, as well as information on using the facilities of the operating system to create multiple phases and to execute them, can be found in the chapter "Calling and Called Programs."

TYPES OF JOBS

A typical job falls into one of several categories. A brief description of these categories follows; a complete discussion is found in the chapter "Preparing COBOL Programs for Processing."

Compile-Only: This type of job involves only the execution of the COBOL compiler. It is useful when checking for errors in COBOL source statements. A compile-only job is also used to produce a module that is to be further processed in a subsequent job.

A compile-only job can consist of one job step or several successive job steps.

Edit-Only: This type of job involves only the execution of the Linkage Editor. It is used primarily to combine modules produced in previous compile-only jobs, and to check that all cross references between modules have been resolved. The programmer can specify that all modules be combined to form one phase; or he can specify that some modules form one phase and that others form additional phases. The phase output produced as the result of an edit-only job can be retained for execution in a subsequent job.

Compile and Edit: This type of job combines the functions of the compile-only and the edit-only jobs. It requires the execution of both the COBOL compiler and the Linkage Editor. The job can include one or more compilations, resulting in one or more modules. The programmer can specify that the Linkage Editor process any or all of the modules just produced; in addition, he can specify that one or more previously produced modules be included in the linkage editor processing.

Execute-Only: This type of job involves the execution of a phase (or multiple phases) produced in a previous job. Once a COBOL program has been compiled and edited successfully, it can be retained as one or more phases and executed whenever needed. This eliminates the need for recompiling and re-editing every time a COBOL program is to be executed.

Edit and Execute: This type of job combines the functions of the edit-only and the execute-only jobs. It requires the execution of both the Linkage Editor and the resulting phase(s).

Compile, Edit, and Execute: This type of job combines the functions of the compile and edit and the execute-only jobs. It calls for the execution of the COBOL compiler, the Linkage Editor, and the problem program; that is, the COBOL program is to be completely processed.

When considering the definition of his job, the programmer should be aware of the following: _if a job step is cancelled during execution, the entire job is terminated; any remaining job steps are skipped_. Thus, in a compile-edit-and execute job, a failure in compilation precludes the editing of the module(s) and phase execution. Similarly, a failure in editing precludes phase execution.

For this reason, a job usually should (but need not) consist of related job steps only. For example, if two independent single-phase executions are included in one job, the failure of the first phase execution precludes the execution of the second phase. Defining each phase execution as a separate job would prevent this from happening. If successful execution of both phases can be guaranteed before the job is run, however, the programmer may prefer to include both executions in a single job.

JOB DEFINITION STATEMENTS

Once the programmer has decided the work to be done within his job and how many job steps are required to perform the job, he can then define his job by writing job control statements. Since these statements are usually punched in cards, the set of job control statements is referred to as a job deck. In addition to job control statements, the job deck can include input data for a program that is executed during a job step. For example, input data for the COBOL compiler -- the COBOL program to be compiled -- can be placed in the job deck.

The inclusion of input data in the job deck depends upon the manner in which the installation has assigned input/output devices. Job control statements are read from the unit named SYSRDR (system reader), which can be either a card reader, a magnetic tape unit, or a disk extent. Input to the processing programs is read from the unit named SYSIPT (system input), which also can be either a card reader, a magnetic tape unit, or a disk extent. The installation has the option of assigning either two separate devices for these units (one device for SYSRDR, a second device for SYSIPT) or one device to serve as both SYSRDR and SYSIPT. If two devices have been assigned, the job deck must consist of only job control statements; input data must be kept separate. If only one device has been assigned, input data must be included within the job deck.

There are four job control statements that are used for job definition: the JOB statement, the EXEC statement, the end-of-data statement (/*), and the end-of-job statement (/&). In this chapter, the discussion of these job control statements is limited to the function and use of each statement. The rules for writing each statement are given in the chapter "Preparing COBOL Programs for Processing."

The JOB statement indicates the beginning of control information for a job. The specified job name is stored in the communications region of the corresponding partition and is used by job accounting and to identify listings produced during execution of the job.

The JOB statement may be omitted, in which case the job name NONAME is stored in the communications region. If the JOB statement is present, it must contain a job name; otherwise, an error condition occurs.

The JOB statement is always printed in positions 1 through 72 on SYSLST and SYSLOG. The time-of-day and date are also printed. The JOB statement causes a skip to a new page before printing is started on SYSLST.

When a JOB statement is encountered, the job control program stores the job name from the JOB statement into the communications region. If the /& statement was omitted, the next JOB statement will cause control to be transferred to the end-of-job routine to simulate the /& statement.

The EXEC statement requests the execution of a program. Therefore, one EXEC statement is required for each job step within a job. The EXEC statement indicates the program that is to be executed (for example, the COBOL compiler, the Linkage Editor). As soon as the EXEC statement has been processed, the program indicated by the statement begins execution.

The end-of-data statement, also referred to as the /* (slash asterisk) statement, defines the end of a program's input data. When the data is included within the job deck (that is, SYSIPT and SYSRDR are the same device), the /* statement immediately follows the input data. For example, COBOL source statements would be placed immediately after the EXEC statement for the COBOL compiler; a /* statement would follow the last COBOL source statement.

Note: For an input file on a 5425 MFCU, the /* card must be followed by a blank card.

When input data is kept separate (that is, SYSIPT and SYSRDR are separate devices), the /* statement immediately follows each set of input data on SYSIPT. For example, if a job consists of two compilation job steps, an editing job step, and an execution job step, SYSIPT would contain the source statements for the first compilation followed by a /* statement, the source statements for the second compilation followed by a /* statement, any input data for the Linkage Editor followed by a /* statement, and perhaps some input data for the problem program followed by a /* statement.

The end-of-job statement, also referred to as the /& (slash ampersand) statement, defines the end of the job. A /& statement must appear as the last statement in the job deck.

OTHER JOB CONTROL STATEMENTS

The four job definition statements form the framework of the job deck. There are a number of other job control statements in the job control language; however, not all of them must appear in the job deck. The job control statements are summarized briefly in Table 1.

The double slash preceding each statement name identifies the statement as a job control statement. Most of the statements are used for data management -- creating, manipulating, and keeping track of data files. (Data files are externally stored collections of data from which data is read and onto which data is written.)

Table 1. Job Control Statements

Statement	Function
// ASSGN	Input/output assignments.
// CLOSE	Closes a logical unit assigned to magnetic tape.
// DATE	Provides a date for the Communication Region.
// DLAB	Disk file label information.
// DLBL	Disk file label information and VSAM file processing.
// EXEC	Execute program.
// EXTENT	Disk file extent.
// JOB	Beginning of control information for a job.
// LBLTYP	Reserves storage for label information.
// LISTIO	Lists input/output assignments.
// MTC	Controls operations on magnetic tape.
// OPTION	Specifies one or more job control options.
// PAUSE	Creates a pause for operator intervention.
// RESET	Resets input/output assignments to standard assignments.
// RSTRT	Restarts a checkpointed program.
// TLBL	Tape label information.
// TPLAB	Tape label information.
// UPSI	Sets user-program switches.
// VOL	Disk/tape label information.
// XTENT	Disk file extent.
// ZONE	Sets the zone for the date.
/*	End-of-data-file or end-of-job-step.
/&	End-of-job.
*	Comments.

This chapter describes in greater detail the three types of job steps involved in processing a COBOL program. Once the reader becomes familiar with the information presented here, he should be able to write control statements by referring only to the next chapter, "Preparing COBOL Programs for Processing."

COMPILATION

Compilation is the execution of the COBOL compiler. The programmer requests compilation by placing in the job deck an EXEC statement that contains the program name FCOBOL, the name of the DOS/VS COBOL compiler. This is the EXEC FCOBOL statement. If the compiler is loaded from a user program, that program must be a cataloged phase. The name of the phase must have as its first four characters 'FCOB'.

Input to the compiler is a set of COBOL source statements, consisting of either a main program or a subprogram. Source statements must be punched in Extended Binary-Coded-Decimal Interchange Code (EBCDIC). The COBOL source statements are read from SYSIPT. The job deck is read from SYSRDR. If SYSRDR and SYSIPT are assigned to the same unit, the COBOL source statements should be placed after the EXEC FCOBOL statement in the job deck.

Output from the COBOL compiler is dependent upon the options specified when the system is generated. This output may include a listing of source statements exactly as they appear in the input deck. The source listing is produced on SYSLST. In addition, the module produced by the compiler may be written on SYSLNK, the linkage editor input unit, and punched on SYSPCH. Separate Data and/or Procedure Division maps, a symbolic cross-reference list, and diagnostic messages can also be produced. The format of compiler output is discussed and illustrated in the chapter "Interpreting Output."

The programmer can override any of the compiler options specified when the system was generated, or include some not previously specified, by using the OPTION control statement in the compile job step. Compiler options are discussed in detail in the chapter "Preparing COBOL Programs for Processing."

EDITING

Editing is the execution of the Linkage Editor. The programmer requests editing by placing in the job deck an EXEC statement that contains the program name LNKEDT, the name of the Linkage Editor. This is the EXEC LNKEDT statement.

Input to the Linkage Editor consists of a set of linkage editor control statements and one or more modules to be edited. These modules include any of the following:

1. Modules that were compiled previously in the job and placed at that time on the linkage editor input unit, SYSLNK.

2. Modules that were compiled in a previous job and saved as module decks. The module decks must be placed on SYSIPT. Linkage editor control statements are read from SYSRDR.

3. Modules that were compiled in a previous job step and cataloged in the relocatable library. The relocatable library is a collection of frequently used routines in the form of modules, that can be included in a program phase via the INCLUDE control statement in the linkage editor job step.

Output from the Linkage Editor consists of one or more phases. A phase may be an entire program or it may be part of an overlay structure (multiple phases).

A phase produced by the Linkage Editor can be executed immediately after it is produced (that is, in the job step immediately following the linkage editor job step), or it can be executed later, either in a subsequent job step of the same job or in a subsequent job. In either of the latter cases, the phase to be executed must be cataloged in the core image libary. Such a phase can be retrieved in the execute job step by specifying the phase name in the EXEC statement, where phase name is the name under which it was cataloged. Otherwise, the phase output is retained only for the duration of one job step following the linkage editor job step. That is, if the module that was just link edited is to be executed in the next job step, it need not have been cataloged. An EXEC statement will cause the phase to be brought in from the temporary part of the core image library and will begin execution. However, the next time such a module is to be executed, the linkage editor job step is required since the phase was not cataloged in the core image library.

If a private core image library is assigned, output from the Linkage Editor is placed in the private core image library (either permanently or temporarily) rather than in the resident system core image library. When execution of a program is requested and a private core image library is assigned, this library is searched first for the requested phase name and then the system core image library is searched.

In addition to the phase, the Linkage Editor produces a phase map on SYSLST. Linkage editor diagnostic messages are also printed on SYSLST. If the NOMAP option of the linkage editor ACTION control statement is specified, no phase map is produced and linkage editor diagnostic messages are listed on SYSLST, if assigned. Otherwise, the diagnostic messages are listed on SYSLOG. The contents of the phase map are discussed and illustrated in the chapter "Interpreting Output."

Linkage editor control statements direct the execution of the Linkage Editor. Together with any module decks to be processed, they form the linkage editor input deck, which is read by the Job Control Processor from SYSIPT and written on SYSLNK.

There are four linkage editor control statements: the ACTION statement, the PHASE statement, the ENTRY statement, and the INCLUDE statement. These statements are discussed in the next chapter.

PHASE EXECUTION

Phase execution is the execution of the problem program, for example, the program written by the COBOL programmer. If the program is an overlay structure (multiple phase), the execution job step actually involves the execution of all the phases in the program.

The phase(s) to be executed must be contained in the core image library. The core image library is a collection of executable phases from which programs are loaded by the Supervisor. A phase is written in the temporary part of the core image library by the Linkage Editor at the time the phase is produced. It is permanently retained (cataloged) in the core image library, if the programmer has so requested, via the CATAL option in the OPTION control statement.

The programmer requests the execution of a phase by placing in the job deck an EXEC statement that specifies the name of the phase. However, if the phase to be executed was produced in the immediately preceding job step, it is not necessary to specify its name in the EXEC statement.

MULTIPHASE PROGRAMS

A COBOL program can be executed as a single phase as long as there is an area of virtual storage available to accommodate it. This area, known as the problem program area, must be large enough to contain the main program and all called subprograms. When a program is too large to be executed as a single phase, it must be structured as a multiphase program.

The overlay structure available to the COBOL programmer for multiphase programs is known as root phase overlay, and is used primarily for programs of three or more phases. One phase of the program is designated as the root phase (main program) and, as such, remains in the problem program area throughout the execution of the entire program. The other phases in the program -- subordinate phases -- are loaded into the problem program area as they are needed. A subordinate phase may overlay any previously loaded subordinate phase, but no subordinate phase may overlay the root phase. One or more subordinate phases can reside simultaneously in storage with the root phase.

Use of the linkage editor control statements needed to effect overlay are discussed in the chapter "Calling and Called Programs."

This chapter provides information about preparing COBOL source programs for compilation, link editing, and execution.

ASSIGNMENT OF INPUT/OUTPUT DEVICES

Almost all COBOL programs include input/output statements calling for data to be read from or written into data files stored on external devices. COBOL programs do not reference input/output devices by their actual physical address, but rather by their symbolic names. Thus, a COBOL program is dependent on the device type but not on the actual device address. Using VSAM, it is not even dependent on the device type. The COBOL programmer need only select the symbolic name of a device from a fixed set of symbolic names. At execution time, as a job control function, the symbolic name is associated with an actual physical device. The standard assignment of physical addresses to symbolic names may be made at system generation time. However, job control statements and operator commands can alter the standard device assignment before program execution. This is discussed later in this chapter.

Using DOS/VS, a logical unit may also be assigned to another logical unit or a general device class or specific device type. For more information on this, see _DOS/VS System Management Guide_ and _DOS/VS System Control Statements_.

The symbolic names are divided into two classes: _system logical units_ and _programmer logical units_.

The system logical units are used by the control program and by IBM-supplied processing programs. SYSIPT, SYSLST, SYSPCH, and SYSLOG can be implicitly referenced by certain COBOL procedural statements. Two additional names, SYSIN and SYSOUT, are defined for background program assignments. The names are valid only to the Job Control Processor, and cannot be referenced in the COBOL program. SYSIN can be used when SYSRDR and SYSIPT are the same device; SYSOUT must be used when SYSLST and SYSPCH are assigned to the same magnetic tape unit. A complete discussion of the assignment of the logical unit SYSCLB can be found in the publication _DOS/VS System Control Statements_.

Programmer logical units are those in the range SYS000 through SYS240 (depending on the number of partitions in the system) and are referred to in the COBOL source language ASSIGN clause.

A COBOL programmer uses the source language ASSIGN clause to assign a file used by his problem program to the appropriate symbolic name. Although symbolic names may be assigned to physical devices at system generation time, the programmer may alter these assignments at execution time by means of the ASSGN control statement. However, if the programmer wishes to use the assignments made at system generation time for his own data files in the COBOL program, ASSGN control statements are unnecessary.

Table 2 is a complete list of symbolic names and their usage.

Table 2. Symbolic Names, Functions, and Permissible Device Types

Symbolic Name	Function	Permissible Device Types
SYSRDR	Input unit for control statements or commands.	Card reader Magnetic Tape unit Disk extent 3540 diskette
SYSIPT	Input unit for programs.	Card reader Magnetic tape unit Disk extent 3540 diskette
SYSPCH	Main unit for punched output.	Card punch Magnetic tape unit Disk extent 3540 diskette
SYSLST	Main unit for printed output.	Printer Magnetic tape unit Disk extent 3540 diskette
SYSLOG	Receives operator messages and logs in job control statements.	Printer keyboard Printer Display operator console
SYSLNK	Input to the Linkage Editor.	Disk extent
SYSRES	Contains the operating system, the core image library, relocatable library, source statement library, and procedure library.	Disk extent
SYSCLB	A private core image library.	Disk extent
SYSSLB	A private source statement library.	Disk extent
SYSRLB	A private relocatable library.	Disk extent
SYSIN	Must be used when SYSRDR and SYSIPT are assigned to the same disk extent. May be used when they are same disk extent. May be used when they are assigned to the same card reader or magnetic tape.	Disk Magnetic tape unit Card reader 3540 Diskette
SYSOUT	This name must be used when SYSPCH and SYSLST are assigned to the same magnetic tape unit. It must be assigned by the operator ASSGN command.	Magnetic tape unit
SYSmax	These units are available to the programmer as work files or for storing data files. They are called <u>programmer logical units</u> as opposed to the above-mentioned names which are always referred to as <u>system logical units</u>. The largest number of programmer logical units available in the system is 240 (SYS000 through SYS240, depending on number of partitions). The value of SYSmax is determined by the distribution of the programmer logical units among the partitions.	Any unit
SYSVIS	Holds virtual storage page data set.	Disk extent
SYSCAT	Holds the VSAM catalog.	Disk extent
SYSREC	Logs error records.	Disk extent

JOB CONTROL

The Job Control Processor for the Disk Operating System/Virtual Storage prepares the system for execution of programs in a batched job environment. Input to the Job Control Processor is in the form of job control statements and job control commands.

JOB CONTROL STATEMENTS

Job control statements are designed for an 80-column punched card format. Although certain restrictions must be observed, the statements are essentially free form. Job control statements conform to these rules:

1. Name. Two slashes (//) identify the statement as a job control statement. They must be in columns 1 and 2. At least one blank immediately follows the second slash.

   Exceptions: The end-of-job statement contains /& in columns 1 and 2; the end-of-data-file statement contains /* in columns 1 and 2; the comment statement contains * in column 1 and a blank in column 2.

2. Operation. This identifies the operation to be performed. It can be up to eight characters long. At least one blank follows its last character.

3. Operand. This may be blank or may contain one or more entries separated by commas. The last term must be followed by a blank, unless its last character is in column 71.

4. Comments. Optional programmer comments must be separated from the operand by at least one space.

Continuation cards are not recognized by the Job Control Processor. For the exception to this rule, see the descriptions of the DLAB and TPLAB statements.

All job control statements are read from the device identified by the symbolic name SYSRDR.

Comments in Job Control Statements

Comment statements (i.e., statements preceded by an asterisk in column 1 followed by a blank) may be placed anywhere in the job deck. The remainder of the card may contain any character from the EBCDIC set. Comment statements are designed for communication with the operator; accordingly, they are written on the console output unit, SYSLOG, in addition to being written on SYSLST. If followed by a PAUSE control statement, the comment statement can be used to request operator action.

Statement Formats

The following notation is used in the statement formats:

1. All upper-case letters represent specifications that are to appear in the actual statement exactly as shown in the statement format. For example, JOB in the operation field of the JOB statement should be punched exactly as shown.

2. All lower-case letters represent generic terms that are to be replaced in the actual statement. For example, jobname is a generic term that should be replaced by the name that the programmer is giving his job.

3. Hyphens are used to join two or more words in order to form a single generic term. For example, device-address is one generic term.

4. Brackets are used to indicate that a specification is optional and is not always required in the statement. For example, [type] indicates that the programmer's replacement for the generic term, type, may or may not appear in the statement, depending on the programmer's requirements.

5. Braces enclosing stacked items indicate that a choice of one item must be made by the programmer. For example:

   ```
   SYS
   PROG
   ALL
   SYSxxx
   ```

   indicates that either SYS, PROG, ALL, or SYSxxx must appear in the actual statement.

6. Brackets enclosing stacked items indicate that a choice of one item may, but need not, be made by the programmer. For example:

```
,X'ss'
,ALT
```

   indicates that either ,X'ss' or ,ALT but not both, may appear in the actual statement, or the specification can be omitted entirely.

7. All punctuation marks shown in the statement formats other than hyphens, brackets, and braces must be punched as shown. This includes periods, commas, and parentheses. For example, ,[date] means that the specification, if present in the statement, should consist of the programmer's replacement for the generic term date preceded by the comma with no intervening space. Even if the date is omitted, the comma must be punched as shown.

8. The ellipsis (...) indicates where repetition may occur at the programmer's option. The portion of the format that may be repeated is determined as follows:

   a. Scanning right to left, determine the bracket or brace delimiter immediately to the left of the ellipsis.

   b. Continue scanning right to left and determine the logically matching bracket or brace delimiter.

   c. The ellipsis applies to the words and punctuation between the pair of delimiters.

Sequence of Job Control Statements

The job deck for a specific job always begins with a JOB statement and ends with a /& (end-of-job) statement. A specific job consists of one or more job steps. The beginning of a job step is indicated by the appearance of an EXEC statement. When an EXEC statement is encountered, it initiates the execution of the job step, which includes all preceding control statements up to, but not including, a previous EXEC statement.

The only limitation on the sequence of statements within a job step is that which is discussed here for the label information statements.

The label statements must be in the order:

```
DLBL
EXTENT (one for each area or file in
    the volume)
```

or

```
TLBL
```

and must immediately precede the EXEC statement to which they apply.

DESCRIPTION AND FORMATS OF JOB CONTROL STATEMENTS

This section contains descriptions and formats of job control statements.

Job control statements, with the exception of /*, /&, and *, contain two slashes in columns 1 and 2 to identify them.

PREP
PGMS

JOB Statement

The JOB control statement indicates the beginning of control information for a job. The JOB control statement is in the following format:

```
// JOB jobname
```

jobname
   is a programmer-defined name consisting of from one to eight alphanumeric characters. Any user comments can appear on the JOB control statement following the jobname (through column 72). The time of day and date appear in columns 73 to 80 when the JOB statement is printed on SYSLST. The time of day and date are also printed in columns 1 through 8 on the next line of SYSLOG.

   If a job is restarted, the jobname must be identical to that used when the checkpoint was taken.

Note: The JOB statement resets the effect of all previously issued OPTION and ASSGN control statements.

ASSGN Statement

The ASSGN control statement assigns a logical input/output unit to a physical device. An ASSGN control statement must be present in the job deck for each data file assigned to an external storage device in the COBOL program where these assignments differ from those established at system generation time. Data files are assigned to programmer logical units in COBOL by means of the source language ASSIGN clause. An ASSGN statement or command can also be used

- to unassign a logical unit to free it for assignment to another partition
- to ignore the assignment of a logical unit, that is, program references to the logical unit are ignored (useful in testing and certain rerun situations)
- to specify an alternate tape unit to be used when the capacity of the original is reached.

The assignment routines check the operands of the ASSGN statement/command for the relationship between the physical device, the logical unit, the type of assignment (permanent or temporary), etc. The following list summarizes the most pertinent items to remember when making assignments:

1. Assignments are effective only for the partition in which they are issued.
2. No physical device except DASD can be assigned to more than one active partition at the same time.
3. All system input and output file assignments to disk or diskette must be permanent.
4. SYSIN must be assigned if both SYSRDR and SYSIPT are to be assigned to the same extent.
5. SYSOUT cannot be assigned to disk or diskette; it must be a permanent assignment if assigned to tape.
6. SYSLNK must be assigned before issuing the LINK or CATAL option in the OPTION statement; otherwise, the option is ignored and the message 'PLEASE ASSIGN SYSLNK' is issued to the operator.
7. If SYSRDR, SYSIPT, SYSLST, or SYSPCH is assigned to tape or diskette, or disk when the system is generated, it will be unassigned by IPL. Such assignments can be made effective only with the job control ASSGN statement or command, because ASSGN also opens the file.
8. Before a tape unit is assigned to SYSLST, SYSPCH, or SYSOUT, all previous assignments to this tape unit must be permanently unassigned. This may be done by using a DVCDN command instead.
9. The assignment of SYSLOG cannot be changed while a foreground partition is active.
10. SYSRES, SYSCAT, and SYSVIS can never be assigned by an ASSGN statement or command. An IPL is required to change these assignments.

The ASSGN control statement may also be used to change a system standard assignment for the duration of the job.

The format of the ASSGN control statement is as follows:

```
                                       ┌,X'ss'┐
 // ASSGN SYSxxx,device-address        │      │
                                       └,ALT  ┘
```

SYSxxx
is one of the logical devices listed in Table 2.

Exception: SYSOUT must be assigned using the ASSGN job control command. Job control commands are described in detail in the publication DOS/VS System Control Statements.

device-address
allows four different formats:

cuu (Rel. 35 and up)
or
X'cuu'
where c is the channel number and uu the unit number in hexadecimal notation. The values of 'cuu' are determined by each installation.

c = 0 for multiplexor channel, 1 through 6 for selector channels 1 through 6.

uu = 00 to FE (0 to 254) in hexadecimal.

UA
indicates that the logical unit is to be unassigned. Any source language input/output operation attempted on this device causes cancellation of the job.

IGN
indicates that the logical unit is to be unassigned. Each time a READ statement for the file assigned to IGN is encountered, control will be transferred to the imperative-statement following the AT END option. The IGN option is not valid for SYSRDR, SYSIPT, and SYSIN. This option is useful in program debugging since source language references to input files residing on symbolic units for which IGN has been specified are ignored. Any file for which the IGN option is used must be a sequential input file. Output files assigned with the IGN option are not supported by DOS/VS COBOL object programs.

X'ss'
is the device specification. It is used for specifying mode settings for 7-track and dual density 9-track tapes. If X'ss' is not specified, the system assumes the value specified at system generation for 7-track tapes and X'C0' for 9-track tapes. The possible specifications for X'ss' are shown in Figure 3.

ALT
must be specified in the control statement that assigns an alternate magnetic tape unit which is used when the capacity of the original assignment is reached. The specifications for the alternate unit must be the same as those of the original unit, since X'ss' cannot be specified. The characteristics of the alternate unit must be the same as those of the original unit. Multiple alternates can be assigned to a symbolic unit.

H1
indicates input hopper one for 2560 or 5425.

H2
indicates input hopper two for 2560 or 5425. H2 may only be assigned to SYSRDR, SYSIPT or SYSPCH.

Device assignments made by the ASSGN control statement are considered temporary. They are in effect until another ASSGN control statement or a RESET statement for that logical unit, or the next /& or JOB statement is read, whichever occurs first. If a RESET, /&, or JOB statement is encountered, the assignment reverts to the standard assignment established at system generation time plus any modification by an ASSGN command.

The COBOL programmer may assign only the programmer logical units (SYS000 through SYS240, depending on the number of partitions) to data files used in his program. For example, if the following ASSIGN clause is used,

```
SELECT IN-FILE ASSIGN TO SYS004-DA-2314-S
```

an ASSGN control statement must appear in the job deck which assigns SYS004 to a physical device if the physical device differs from the permanent assignment. In this case, the physical device must be a 2314 direct access device. An example of such a control statement is:

```
// ASSGN SYS004,X'00C'
```

Physical unit X'00C' was permanently assigned to a 2314 direct access device at system generation time.

Note: The ASSGN control statement is necessary only when the symbolic unit assignment is being made to a physical device address which differs from that established at system generation time.

"Appendix H: Sample Job Decks" contains illustrations of ASSGN statement usage.

		7-Track Tape		
ss	Bytes per Inch	Parity	Translate Feature	Convert Feature
10	200	odd	off	on
20	200	even	off	off
28	200	even	on	off
30	200	odd	off	off
38	200	odd	on	off
50	556	odd	off	on
60	556	even	off	off
68	556	even	on	off
70	556	odd	off	off
78	556	odd	on	off
90	800	odd	off	on
A0	800	even	off	off
A8	800	even	on	off
B0	800	odd	off	off
B8	800	odd	on	off
		9-Track Tape		
C0	800	single density 9-track		
C0	1600	single density 9-track		
C0	1600	dual density 9-track		
C8	800	dual density 9-track		
D0	6250	single density 9-track		
D0	6250	dual density 9-track		

Figure 3. Possible Specifications for X'ss' in the ASSGN Control Statement

CLOSE Statement

The CLOSE control statement is used to close either a system or programmer logical unit assigned to tape. As a result of the CLOSE control statement, a standard end-of-volume label set is written and the tape is rewound and unloaded. The CLOSE statement applies only to a temporarily assigned logical unit, that is, a logical unit for which an ASSGN control statement has been specified within the same job. The format of the CLOSE control statement is as follows:

```
┌──────────────────────────────────────────┐
│                                          │
│                ┌,X'cuu" [,X'ss']┐        │
│                │,UA             │        │
│// CLOSE SYSxxx │,IGN            │        │
│                └,ALT            ┘        │
└──────────────────────────────────────────┘
```

The logical unit can optionally be reassigned to another device, unassigned, or switched to an alternate unit.

Note that when SYSxxx is a system logical unit, one of the optional parameters must be specified. When closing a programmer logical unit, no optional parameter need be specified.

SYSxxx
: may only be used for magnetic tape and may be specified as SYSPCH, SYSLST, SYSOUT, or SYS000 through SYS240, depending on the number of partitions.

| cuu (Rel. 35 and up)
or
X'cuu'
: specifies that after the logical unit is closed, it will be assigned to the channel and unit specified. (See "ASSGN Control Statement" for an explanation of 'cuu'.) When reassigning a system logical unit, the new unit will be opened if it is either a mass storage device or a magnetic tape at load point.

X'ss'
: represents device specification for mode settings on 7-track and 9-track tape. (See "ASSGN Control Statement" for an explanation of 'ss'.) If X'ss' is not specified, the mode settings remain unchanged.

UA
: specifies that the logical unit is to be closed and unassigned.

IGN
: specifies that the logical unit is to be closed and unassigned with the ignore option. This operand is invalid for SYSRDR, SYSIPT, or SYSIN.

ALT
: specifies that the logical unit is to be closed and an alternate unit is to be opened and used. This operand is valid only for system logical output units (SYSPCH, SYSLST, or SYSOUT) currently assigned to a magnetic tape unit.

DATE Statement

The DATE control statement contains a date that is put in the Communication Region of the Supervisor. A complete description of the fields of the Communication Region is given in "Appendix G: Communication Region." The DATE statement is in one of the following formats:

```
┌──────────────────────────────────────────┐
│// DATE mm/dd/yy                          │
├──────────────────────────────────────────┤
│// DATE dd/mm/yy                          │
└──────────────────────────────────────────┘
```

where:
mm = month (01 to 12)
dd = day (01 to 31)
yy = year (00 to 99)

The format to be used is the format selected when the system was generated.

When the DATE statement is used, it applies only to the current job being executed. The Job Control Processor does not check the operand except to ensure that its length is eight characters. If no DATE statement is specified in the current job, the Job Control Processor supplies the date given in the last SET command. The SET command is discussed in detail in the publication DOS/VS System Control Statements.

A DATE statement should be included in every job deck that has as one of its job steps the execution of a COBOL program that utilizes the special register CURRENT-DATE, if the date desired is other than that designated in the previous SET command.

The DATE statement should be used at compile time so that the DATE-COMPILED paragraph is accurate and the WHEN-COMPILED special register is effective.

TLBL Statement

The TLBL control statement replaces the VOL and TPLAB combination used in previous versions of the system. However, the current system will continue to support these statements. The TLBL control statement contains file label information for tape label checking and writing. Its format follows:

```
// TLBL filename,
     [,'file-identifier'][,date]
     [,file-serial-number]
     [,volume-sequence-number]
     [,file-sequence-number]
     [,generation-number]
     [,version-number]
```

filename
: identifies the file to the control program. It can be from three to seven characters in length. If the following SELECT sentence appears in a COBOL program:

```
SELECT NEWFILE ASSIGN TO
SYS003-UT-2400-S-OUTFILE
```

the filename operand on control statements for this file must be OUTFILE. If the SELECT clause were coded:

```
SELECT NEWFILE ASSIGN TO
SYS003-UT-2400-S
```

the filename operand on the control statement for the file must be SYS003.

'file-identifier'
: consists of from 1 to 17 characters, contained within apostrophes, indicating the name associated with the file on the volume. This operand may contain embedded blanks. If this operand is omitted on output files, the <u>filename</u> will be used. If this operand is omitted on input files, no checking will be done.

date
: consists of from one to six characters, in the format yy/ddd, indicating the expiration date of the file for output or the creation date for input. (The day of the year may consist of from one to three characters.) For output files, a one to four character retention period (d-dddd) may be specified. If this operand is omitted, a 0-day retention period will be assumed for output files. For input files, no checking will be done if this operand is omitted or if a retention period is specified.

file-serial-number
: consists of from one to six characters indicating the volume serial number of the first (or only) reel of the file. If fewer than six characters are specified, the field will be right-justified and padded with zeros. If this operand is omitted on output files, the volume serial number of the first (or only) reel of the file will be used. If the operand is omitted on input files, no checking will be done.

volume-sequence-number
: consists of from one to four characters in ascending order for each volume of a multivolume file. This number is incremented automatically by OPEN and CLOSE routines as required. If this operand is omitted on output files, BCD 0001 will be used. If omitted on input files, no checking is done.

file-sequence-number
: consists of from one to four characters in ascending order for each file of a multifile volume. This number is incremented automatically by OPEN and CLOSE routines as required. If this operand is omitted on output files, BCD 0001 will be used. If it is omitted on input files, no checking will be done.

generation-number
: consists of from one to four numeric characters that modify the file-identifier. If this operand is omitted on output files, BCD 0001 is used. If it is omitted on input files, no checking will be done.

version-number
: consists of from one to two numeric characters that modify the generation number. If this operand is omitted on output files, BCD 01 will be used. If it is omitted on input files, no checking will be done.

<u>Note</u>: If a tape file with standard labels is opened two different ways in the same COBOL program, and that file resides on a multifile volume, the programmer should use two separate TLBL cards with different filenames specified on each.

DLBL Statement

The DLBL control statement, in conjunction with the EXTENT statement, replaces the VOL, DLAB, and XTENT combination used in previous versions of the Disk Operating System. The DLBL statement has the following format:

```
// DLBL filename
     ,['file-identifer'],[date],[codes]
     ,[BLKSIZE=n], (CISIZE=n)
```

filename
: identifies the file to the control program. It can be from three to seven characters long. If the following SELECT sentence appears in a COBOL program:

  SELECT INFILE ASSIGN TO
  SYS005-DA-2314-A-INPUTA

  the filename operand on control statements for this file must be INPUTA. If the SELECT sentence is coded:

  SELECT INFILE ASSIGN TO
  SYS005-DA-2314-A

  the filename operand on control statements for the file must be SYS005.

'file-identifier'
: is the name associated with the file on the volume. This can consist of from 1 to 44 alphanumeric characters contained within apostrophes, including the file-identifier and, if used, generation-number and version-number of generation. If fewer than 44 characters are used, the field is left-justified and padded with blanks. If this operand is omitted, _filename_ will be used.

date
: consists of from one to six characters indicating either the retention period of the file in the format d through dddd (0-9999), or the absolute expiration date of the file in the format yy/ddd. When the d through dddd format is used, the file is retained for the number of days specified as dddd. For example, if _date_ is specified as 31, the file will be retained a month from the day of creation. When the yy/ddd format is used, the file is retained until the day (ddd) in the year (yy) specified. For example, if _date_ is specified as 90/200, the file will be retained through the 200th day of the year 1990.

  If _date_ is omitted when the file is created, a 7-day retention period is assumed. If this operand is present for a file opened as INPUT or I-O, it is ignored.

codes
: is a 2 to 4 character field indicating the type of file label, as follows:

SD	=	Sequential Disk
DA	=	Direct Access
ISC	=	Indexed Sequential using Load Create
ISE	=	Indexed Sequential using Load Extension, Add, or Retrieve
DU	=	3540 Diskette
VSAM	=	VSAM file

  If _code_ is omitted, SD is assumed.

BLKSIZE=n
: specifies the number of bytes in a physical record. n must be less than 32,768. This parameter is valid for the 3330-11 and 3350 devices only, and its use is limited to sequential files. If specified, it overrides the BLKSIZE specification in the definition of the file (DTF). It permits reblocking of existing files to a new physical record size when they are transferred to a 3330-11 or 3350 device, without requiring recompilation of the DTF. If the BLKSIZE parameter is not specified in the DLBL statement, the new files are assumed to have the blocksize specified in the DTF. This parameter is not valid for the compiler workfiles.

  For further information, see _DOS/VS System Control Statements_.

CISIZE=n
: specifies the control interval size for SAM files on fixed block devices, and improves space allocation on such devices. The size specified must be a multiple of the value specified in the BLKSIZE=n operand. This operand is valid only for a DLBL statement with the code SD. It is not valid for compiler workfiles.

"Appendix H: Sample Job Decks" contains illustrations of DLBL statement usage.

See the section "Processing 3540 Diskette Unit Files" for the use of DLBL Cards for 3540 and the section "Virtual Storage Access Method" for use of DLBL cards for VSAM.

EXTENT Statement

The EXTENT control statement defines each area (or extent) of a DASD file -- a file assigned to a mass storage device. One or more EXTENT control statements must follow each DLBL statement.

The EXTENT control statement replaces the XTENT statement used in previous versions of the Disk Operating System. For more information on the XTENT statement, see DOS/VS System Control Statements.

The format of the EXTENT control statement is:

```
// EXTENT [symbolic-unit],[serial-number]
    ,[type],[sequence-number]
    ,[relative-track],[number-of-tracks]
    ,[split-cylinder-track],[B=bins]
```

symbolic-unit
: is a 6-character field indicating the symbolic unit (SYSxxx) of the volume for which this extent is effective. If this operand is omitted, the symbolic unit of the preceding EXTENT statement will be used. When specified, symbolic-unit may be any SYSxxx assigned to the device type indicated in the SELECT sentence for the file. For example, if the following coding appears in a COBOL program:

```
SELECT OUTFILE ASSIGN TO
SYS004-DA-2314-A
```

the symbolic unit in the EXTENT control statement can by any SYSxxx assigned to a 2314 disk pack. The symbolic unit operand is not required for an IJSYSxx filename, where xx is IN, PH, LS, RS, SL, or RL. If SYSRDR or SYSIPT is assigned, this operand must be included.

serial-number
: consists of from one to six characters indicating the volume serial number of the volume for which this extent is effective. If fewer than six characters are used, the field will be right-justified and padded with zeros. If this operand is omitted, the volume serial number of the preceding EXTENT control statement will be used. If no serial number was provided in the EXTENT control statement, the serial number will not be checked and it will be the programmer's responsibility if files are destroyed as a result of mounting the incorrect volume.

type
: consists of one character indicating the type of the extent, as follows:

1 -- Data area (no split cylinder)
2 -- Overflow area (for an indexed file)
4 -- Index area (for an indexed file)
8 -- Data area (split cylinder)

If this operand is omitted, 1 is assumed.

sequence-number
: consists of from one to three characters containing a decimal number from 0 to 255 indicating the sequence number of this extent within a multi-extent file. Extent sequence 0 is used for the master index of an indexed file. If the master index is not used, the first extent of an indexed file has the sequence number 1. The extent sequence number for all other types of files begins with 0. If this operand is omitted for the first extent of ISAM files, the extent will not be accepted. For SD or DA files, this operand is not required. For DA files this operand should be specified when using more than one EXTENT for a file. Direct files can have up to five extents. Indexed files can have up to eleven data extents (nine prime, one cylinder index, one separate overflow).

relative-track
: consists of from one to five characters indicating the sequential number of the track, relative to zero, where the data extent is to begin. If this field is omitted on an ISAM file, the extent will not be accepted. This field is not required for DA input or for SD input files (the extents from the file labels will be used).

For fixed block devices, this operand is a number from 2 to 2,147,483,645 that specifies the physical block at which the extent should start.

Formulas for converting actual to relative track addresses (RT) and relative track to actual for the DASD devices follow.

<u>Actual to Relative:</u>

2311	10 x cylinder number + track number = RT
2314 or 2319	20 x cylinder number + track number = RT
2321	1000 x subcell number + 100 x strip number + 20 x block number + track number = RT
3330	19 x cylinder number + track number = RT
3340	12 x cylinder number + track number = RT
3350	30 x cylinder number + track number = RT

Relative to Actual:

2311	RT / 10 =	quotient is cylinder, remainder is track
2314 or 2319	RT / 20 =	quotient is cylinder, remainder is track
3330	RT / 19 =	quotient is cylinder, remainder is track
2321	RT / 1000 =	quotient is subcell, remainder1
	remainder1 / 100 =	quotient is strip, remainder2
	remainder2 / 20 =	quotient is block, remainder is track
3340	RT / 12 =	quotient is cylinder, remainder is track
3350	RT / 30 =	quotient is cylinder, remainder is track

number-of-tracks
: consists of from one to five characters indicating the number of tracks to be allocated to the file. For SD input files, this field may be omitted. The number of tracks for a split cylinder file must be a multiple of the number of cylinders specified for the file and the number of tracks specified for each cylinder.

For fixed block devices, this operand is a number from 1 to 2,147,483,645 that specifies the number of physical blocks in the extent.

split-cylinder-track
: consists of from one to two characters, with a value of 0 through 19, indicating the upper track number for the split cylinder in SD files.

bins
: consists of from one to two characters identifying the 2321 bin that the extent was created for, or on which the extent is currently located. If the field is one character, the creating bin is assumed to be zero. There is no need to specify a creating bin for SD or ISAM files. If this operand is omitted, bin 0 is assumed for both bins. If the operand is included and positional operands are omitted, only one comma is required preceding the keyword operand. If any operands preceding the bin specification are omitted, one comma for each operand is acceptable, but unnecessary.

Figure 4 shows examples of using the DLBL statement in conjunction with the EXTENT statement. "Appendix H: Sample Job Decks" contains illustrations of EXTENT statement usage.

VOL, DLAB, TPLAB AND XTENT STATEMENTS

These statements have been replaced by the DLBL, TLBL, and EXTENT statements, and, although they are still supported by the Disk Operating System, they cannot be used for 3330 or 3340 disk files, or for VSAM files. Details as to their usage can be found in DOS/VS System Control Statements. For their use with respect to COBOL, see IBM DOS Full American National Standard COBOL Programmer's Guide. When new label information statements are prepared, DLBL, TLBL, and EXTENT should be used.

LBLTYP Statement

The LBLTYP control statement defines the amount of storage to be reserved at linkage edit time in the problem program area of storage in order to process tape and nonsequential DASD file labels. It applies to both background and foreground object programs, and is required if the file contains standard labels.

The LBLTYP control statement immediately precedes the // EXEC LNKEDT statement in the job deck, with the exception of self-relocating programs for which it is instead submitted immediately preceding the // EXEC statement for the program. The format of the LBLTYP control statement is:

```
// LBLTYP  { TAPE[(nn)] }
           { NSD(nn)    }
```

Direct file:

The following DLBL and EXTENT statements describe a direct file occupying 840 tracks, beginning on relative track 10.

```
    // DLBL MASTER,,75/001,DA
    // EXTENT SYS015,111111,1,0,10,840
```

Indexed file:

The following DLBL and EXTENT statements describe an indexed file on a 2314 occupying 100 tracks, beginning on relative track 1100. The first EXTENT allocates a 20-track cylinder index. The second EXTENT allocates a 80-track data area.

```
    // DLBL MASTER,,75/001,ISC
    // EXTENT SYS015,111111,4,1,1100,20
    // EXTENT SYS015,111111,1,2,1120,80
```

Figure 4. Sample Label and File Extent Information for Mass Storage Files

TAPE[(nn)]
is used only if tape files requiring label information are to be processed and if no nonsequential DASD files are to be processed. nn is optional and is present only for future expansion. It is ignored by the Job Control Processor.

NSD(nn)
is used if any nonsequential DASD files are to be processed, regardless of other type files that are used. nn specifies the largest number of extents to be used for a single file.

LISTIO Statement

The LISTIO control statement causes the system to print a list of input/output assignments on SYSLST. The format of the LISTIO control statement is:

```
// LISTIO  { SYS
             PROG
             BG
             F1
             F2
             F3
             F4
             ALL
             SYSxxx
             UNITS
             DOWN
             UA
             cuu
             X'cuu'
             ASSGN (Rel. 35 and up) }
```

SYS
causes the physical units assigned to all system logical units to be listed.

PROG
causes the physical units assigned to all background programmer logical units to be listed.

BG
lists the physical units assigned to all logical units of the background partition.

F1
causes the physical units assigned to all foreground-one logical units to be listed.

F2
causes the physical units assigned to all foreground-two logical units to be listed.

F3
causes the physical units assigned to all foreground-three logical units to be listed.

F4
causes the physical units assigned to all foreground-four logical units to be listed.

ALL
causes the physical units assigned to all logical units to be listed.

SYSxxx
causes the physical units assigned to the logical unit specified to be listed.

UNITS
causes the logical units assigned to all physical units to be listed.

DOWN
causes all physical units specified as inoperative to be listed.

UA
causes all physical units not currently assigned to a logical unit to be listed.

cuu (Release 35 and up)
or
X'cuu'
causes the logical units assigned to the physical unit specified to be listed.

ASSGN
causes all system and program logical units assigned to the current partition to be listed.

MTC Statement

The MTC control statement controls 2400 and 3400 series magnetic tape operations. The format is as follows:

```
// MTC opcode, {SYSxxx} [,nn]
               {X'cuu' }
               {cuu    }
```

opcode
specifies the operation to be performed. opcode can be chosen from the following:

BSF -- Backspace to tapemark

BSR -- Backspace to interrecord gap

ERG -- Erase gap (write blank tape)

FSF -- Forward space to tapemark

FSR -- Forward space to interrecord gap

RUN -- Rewind and unload

REW -- Rewind

WTM -- Write tapemark

SYSxxx
: represents any logical unit assigned to magnetic tape upon which the MTC control statement is to operate.

X'cuu'
: represents any physical unit assigned to magnetic tape upon which the MTC control statement is to operate.

[,nn]
: is the decimal number (01 through 99) which, if specified, represents the number of times the operation is to be performed. If nn is omitted, the operation is performed once.

OPTION Statement

The OPTION control statement is used to specify one or more of the options of the Job Control Processor. The format of the OPTION statement is:

```
┌──────────────────────────────────────────┐
|// OPTION option1[,option2]...            |
└──────────────────────────────────────────┘
```

The order in which the selected options appear in the operand field is arbitrary. Options are reset to the standard established at system generation time upon encountering the next JOB statement or the /& statement.

The options are:

LOG
: causes the listing of columns 1 through 80 of all control statements on SYSLST. If LOG is not the standard established at system generation time, control statements are not listed until a LOG option is encountered. Once a LOG option statement is read, logging continues from job step to job step until a NOLOG option is encountered or until either the JOB or /& control statement is encountered.

NOLOG
: suppresses the listing of all control statements on SYSLST until a LOG option is encountered, or until either the JOB or /& control statement is encountered.

DUMP
: causes a dump of the registers and virtual storage to be printed on SYSLST in the case of an abnormal program termination (such as a program check). Using the compiler SYMDMP, FLOW, or STATE features, it may not be necessary to use this option.

NODUMP
: suppresses the DUMP option.

LINK
: indicates that the object module is to be link edited. When the LINK option is used, the output of the COBOL compiler is written on SYSLNK. The LINK option must always precede an EXEC LNKEDT statement in the job deck. (CATAL also causes the LINK option to be set.) LINK is not acceptable to the Job Control Processor operating in the foreground unless the private core image library option is supported and a private core image library is assigned.

NOLINK
: suppresses the LINK option. The COBOL compiler can also suppress the LINK option if the program contains an error that would preclude the successful execution of the program, or if SYNTAX is in effect, or if CSYNTAX is in effect and an E-level error is encountered.

DECK
: causes the COBOL compiler to punch an object module on SYSPCH. If both DECK and LINK are specified, the output of the compiler is written on both SYSPCH and SYSLNK.[1]

NODECK
: suppresses the DECK option. The DECK option is also suppressed if SYNTAX is in effect, or if CSYNTAX is in effect and E-level errors exist.

LIST
: causes the compiler to write the COBOL source statements on SYSLST. If lister is in effect, the LIST option is overridden; LISTER causes a listing regardless of whether LIST or NOLIST is specified.

NOLIST
: suppresses the LIST option.

LISTX
: causes the COBOL compiler to write a Procedure Division map on SYSLST. In addition, global tables, literal pools, register assignments, and procedure block assignments will be provided. You may want to use the CBL

----

[1]The //option card options pertaining to the compiler will be suppressed if the "LISTER ONLY" option of lister is in effect. Otherwise, when "LISTER AND COMPILE" is in effect, the options specified will be in effect for compilation.

option CLIST (condensed list) in place of this.[1]

NOLISTX
suppresses the LISTX option, as do the same conditions as cause DECK to be suppressed.

XREF
causes the COBOL compiler to write a symbolic cross-reference list on SYSLST. You may want to use the CBL option SXREF in place of this, or the lister cross-reference information for large COBOL programs.

NOXREF
suppresses the XREF option. SXREF also suppresses XREF, as do the same conditions as cause DECK to be suppressed.

SYM
causes the COBOL compiler to write a Data Division map on SYSLST. In addition, global tables, literal pools, register assignments, and procedure block assignments will be provided.[1]

NOSYM
suppresses the SYM option.

ERRS
causes the COBOL compiler to write the diagnostic messages related to the source program on SYSLST.[1]

NOERRS
suppresses the ERRS option. It does not suppress FIPS messages.

CATAL
causes the cataloging of a phase or program in the core image library upon completion of a linkage editor job step. CATAL also causes the LINK option to be set. CATAL is not accepted by the Job Control Processor operating in a batched-job foreground environment unless the private core image library option is supported and a private core image library is assigned.

STDLABEL
causes the standard label track to be cleared and all DASD or tape labels submitted after this point to be written on the standard label track. This option is reset to the USRLABEL option at end-of-job or end-of-job step. All file definition statements submitted after the STDLABEL option are available to any program in any area until another set of standard file definition statements is submitted. STDLABEL is not accepted by the Job Control Processor operating in a batched-job foreground environment. All file definition statements following OPTION STDLABEL are included in the standard file definition set until one of the following occurs:

- End-of-job step
- End-of-job
- OPTION USRLABEL is specified
- OPTION PARSTD is specified

USRLABEL
causes all DASD or tape labels submitted after this point to be written at the beginning of the user label track.

PARSTD
causes all DASD or tape labels submitted after this point to be written at the beginning of the partition standard label track. The PARSTD option is reset to the USRLABEL option at end-of-job or end-of-job step. All file definition statements submitted after the PARSTD option will be available to any program in the current partition until another set of partition standard file definition statements is submitted. All file definition statements submitted after OPTION PARSTD will be included in the standard file definition set until one of the following occurs:

- End-of-job step
- End-of-job
- OPTION USRLABEL is specified
- OPTION STDLABEL is specified

For a given filename, the sequence of search for label information during an OPEN is the USRLABEL area, followed by the PARSTD area, followed by the STDLABEL area.

Note: If NOLINK and NODECK are requested on the OPTION control statement and either SYMDMP or OPT is specified on the CBL card, the SYMDMP or OPT specification is ignored.

----
[1]The //option card options pertaining to the compiler will be suppressed if the "LISTER ONLY" option of lister is in effect. Otherwise, when "LISTER AND COMPILE" is in effect, the options specified will be in effect for compilation.

The options specified in the OPTION statement remain in effect until a contradictory option is encountered or until a JOB control statement is read. In the latter case, the options are reset to the standard that was established at system generation time.

Any assignment for SYSLNK, after the occurrence of the OPTION statement, cancels the LINK and CATAL options. These two options are also canceled after each occurrence of an EXEC statement with a blank operand.

PAUSE Statement

The PAUSE control statement allows for operator intervention between job steps. The format of the PAUSE control statement is:

```
// PAUSE [comments]
```

The PAUSE control statement is effective just before the next input control statement in the job deck is read. The PAUSE control statement always prints on SYSLOG and SYSLST.

An example of this statement is:

```
// PAUSE SAVE SYS004, SYS005, MOUNT
     NEW TAPES
```

This sample statement instructs the operator to save the output tapes and mount two new tapes.

When the PAUSE statement is encountered by the Job Control Processor, processing is stopped in the partition until a response is given. The end/enter key causes processing to continue.

RESET Statement

The RESET control statement resets input/output assignments to the standard assignments. The standard assignments are those specified at system generation time plus any modifications made by the operator by means of the ASSGN command without the TEMP option. The RESET command is discussed in detail in the publication DOS/VS System Control Statements. The format of the RESET statement is:

```
// RESET  { SYS    }
          { PROG   }
          { ALL    }
          { SYSxxx }
```

SYS
: resets all system logical units to their standard assignments.

PROG
: resets all programmer logical units to their standard assignments.

ALL
: resets all system and programmer logical units to their standard assignments.

SYSxxx
: resets the logical unit specified to its standard assignment.

RSTRT Statement

A restart facility is available for checkpoint programs. A programmer can use the source language RERUN clause in his program to cause checkpoint records to be written. This allows sufficient information to be stored so that program execution can be restarted at a specified point. The checkpoint information includes the registers, tape positioning information, a dump of virtual storage, and a restart address.

The restart facility allows the programmer to continue execution of an interrupted job at a point other than the beginning. The procedure is to submit a group of job control statements including a RSTRT control statement. The format is as follows:

```
// RSTRT SYSxxx,nnnn[,filename]
```

SYSxxx
: is the symbolic unit name of the 2400, 3410, 3420, 2311, 2314, 2319, 3330, 3340, 3350, or fixed block devices checkpoint file used for restarting. This unit must have been assigned previously.

nnnn
: is the identification of the checkpoint record to be used for restarting. This serial number consists of four characters. It corresponds to the checkpoint identification used when the checkpoint was taken. The serial number is supplied by the checkpoint routine.

filename
: is the symbolic name of the disk checkpoint file used for restarting. It must be identical to the SYSxxx of the system-name specified in the RERUN clause.

When a checkpoint is taken, the completed checkpoint is noted on SYSLOG. Restarting can be done from any checkpoint record, not just the last. The jobname specified in the JOB statement must be identical to the jobname used when the checkpoint was taken. The proper input/output device assignments must precede the RSTRT control statement.

Assignment of input/output devices to symbolic unit names may vary from the initial assignment. Assignments are made for restarting jobs in the same manner as assignments are made for normal jobs.

See the chapter "Program Checkout" for further details on taking checkpoints and restarting a program for which checkpoints have been taken.

UPSI Statement

The UPSI control statement allows the programmer to set program switches that can be tested by problem programs at execution time. The UPSI control statement has the following format:

```
// UPSI nnnnnnnn
```

nnnnnnnn
: consists of from one to eight characters of 0, 1, or X. Positions containing 1 are set to 1; positions containing X are unchanged. Unspecified rightmost positions are assumed to be X.

The UPSI byte is the 24th byte in the Communication Region of the Supervisor. A complete description of the fields of the Communication Region is given in "Appendix G: Communication Region." The Job Control Processor clears the UPSI byte to binary zeros before reading control statements for each job. When the UPSI control statement is read, the Job Control Processor sets these bits to the programmer's specifications. Any combination of the eight bits can be tested in the COBOL source program at execution time by means of the source language switches UPSI-0 through UPSI-7.

EXEC Statement

The EXEC statement (Execute Program or Procedure) indicates the end of control information for a job step and the beginning of execution of a program, in which case it must be the last command or statement processed before a job step is executed.

```
// EXEC [[PGM=]programname][,REAL][,SIZE]
        [PROC=procedurename]
```

PGM=programname
: represents the name of the program in the core image library to be executed. The program name corresponds to the first or only phase of the program in the library. The program name can be one to eight alphameric characters (0-9, A-Z, #, $, @). The first character must not be numeric.

  If the program to be executed has just been processed by the linkage editor, the program name is omitted and the PGM keyword cannot be used.

REAL
: indicates that the job step started by EXEC will be executed in real mode. If REAL is not specified the job step is always executed in virtual mode. REAL cannot be specified for programs using VSAM, the 3886, for ISAM programs using the ISAM interface program or, for programs compiled with the CBL option count.

SIZE=size
: Size can be nK, AUTO or (AUTO, nK).

(a) If specified with REAL, it indicates the size of that part of the real partition that will be needed by the job step's associated EXEC. The remaining part of the real partition is given to the page pool.

If SIZE is omitted and REAL is specified, the whole real partition is used by the job step.

In Release 35 (DOS/VS) and up, if the COBOL compiler is executed in a real partition, a SIZE parameter must be specified. Also, make sure there is enough real GETVIS space available.

(b) If used without REAL, it specifies that the virtual partition to be used by the job step is divided into two parts: the lower part with a size of nK will contain the program initiated with EXEC; the upper part serves as additional storage pool for other modules (for example, VSAM) required by the program in that partition. The program reserves the upper storage part for its needs by issuing GETVIS macros with the required amount of storage as parameter; it releases the storage by issuing FREEVIS macros.

If SIZE is omitted, the whole virtual partition is used for the job initiated with EXEC.

SIZE (without REAL) must always be specified for VSAM programs or for ISAM programs using the ISAM Interface Program (IIP), as well as for 3886 processing, and for programs compiled with the CBL option count.

If you specify SIZE=AUTO, the system automatically uses the information in the core image directory to calculate the size of the program to be loaded. If you specify SIZE=(AUTO,nK). The system adds nK bytes to the calculated length.

The following restrictions apply to n:

- n must not be larger than the size of the partition it refers to.
- n must be greater than zero.
- if n is not a multiple of 2, n+1 is used

Note: If you specify SIZE=AUTO, a part of the partition is allocated to the page pool. The storage space left is not sufficient for the compiler program. Thus you should not specify SIZE=AUTO in an EXEC FCOBOL statement (for more detailed information, refer to System Control Statements).

Note: If CBL option SYMDMP is used, see Appendix F: "System and Size Considerations."

PROC=procedurename
represents the name of the procedure to be retrieved from the procedure library. The procedure name can be from one to eight alphanumeric characters, the first of which must be alphabetic.

For more information on cataloged procedures, as well as the use of overwrite statements and the rules that apply to temporary procedure modification, refer to the DOS/VS System Management Guide and the chapter "Librarian Functions" in this book.

CBL STATEMENT -- COBOL OPTION CONTROL CARD

Although some options for compilation are specified either at system generation time or in the OPTION control statement, the COBOL compiler provides an additional statement, the CBL statement, for the specification of compile-time options unique to COBOL.

The CBL card must be placed between the EXEC FCOBOL statement and the first statement in the COBOL program. The CBL card cannot be continued. However, if specification of options will continue past column 71, multiple CBL cards may be used.

The options shown in the following format may appear in any order. No comments should appear in the operand field. Underscoring indicates the default case. To change the defaults for your installation, see "Changing the Installation Defaults."

```
CBL [BUF=nnnnn] [,SEQ    ] [,FLAGW]
                [,NOSEQ  ] [,FLAGE]

 [,SUPMAP  ]   [,SPACEn]  [,CLIST  ]
 [,NOSUPMAP]              [,NOCLIST]

[,STXIT  ] [,QUOTE] [,TRUNC  ] [,ZWB  ]
[,NOSTXIT] [,APOST] [,NOTRUNC] [,NOZWB]

[,SXREF  ]   [,PMAP=h]     [,OPTIMIZE  ]
[,NOSXREF]                 [,NOOPTIMIZE]
                           [,OPT       ]
                           [,NOOPT     ]

 [,FLOW[=nn]] [,STATE  ]   [,SYNTAX  ]
              [,NOSTATE]   [,CSYNTAX ]
                           [,NOSYNTAX]

 [,SYMDMP[=filename]] [,VERBSUM  ]
                      [,NOVERBSUM]

[,VERBREF  ] [,COUNT  ]
[,NOVERBREF] [,NOCOUNT]

                                     [      {A}]
                                     [      {B}]
[,CATALR  ] [,LIB  ] [,VERB  ]       [,LVL= {C}]
[,NOCATALR] [,NOLIB] [,NOVERB]       [      {D}]
                                     [,NOLVL   ]
```

CBL
: must begin in column 2 (column 1 must be blank) and be followed by at least one blank.

BUF=nnnnn
: the BUF option specifies the amount of storage to be assigned to each compiler work file buffer. _nnnnn_ is a decimal number from 512 to 32,767. If this option is not specified, 512 is assumed. The BUF option should be used to specify an optional blocksize (which will depend on the device type) for the workfiles. Usually, a larger blocksize will enhance the performance of the compiler. However, for any given BUF specification, the compiler space requirements (over 64K) are increased by a factor of 6x(nnnnnn-512). 6x(nnnnnn-512K)+64K =partition size.

SEQ
NOSEQ
: indicates whether or not the compiler is to check the sequence of source statements. If SEQ is specified and a statement is not in sequence, it is flagged. If the lister feature is invoked, the source statements are resequenced automatically before the sequence check is performed.

FLAGW
FLAGE
: determines which diagnostics the compiler will list. FLAGW indicates that all diagnostics will be listed (severity levels W, C, E, and D). FLAGE indicates that only those diagnostics with severity levels C, E, and D will be listed. This has no effect on FIPS messages.

SUPMAP
NOSUPMAP
: causes the CLIST and LISTX options to be suppressed if an E-level diagnostic message is produced by the compiler. For the DECK option, refer to OBJECT MODULE in the chapter "Interpreting Output."

SPACEn
: indicates the type of spacing to be used on the output listing. _n_ can be specified as either 1 (single spacing), 2 (double spacing), or 3 (triple spacing). If the SPACEn option is omitted, single spacing is provided. Single spacing is always in effect if the lister feature is invoked.

CLIST
NOCLIST
: indicates that a condensed listing is to be produced. The condensed listing will contain only the address of the first generated instruction for each verb in the Procedure Division. In addition, global tables, literal pools, register assignments, and procedure block assignments will be provided. The CLIST option overrides the LISTX or NOLISTX options. The LISTX or NOLISTX options are either established at system generation time or specified in the OPTION control statement.

STXIT
NOSTXIT
: enables a USE AFTER STANDARD ERROR declarative to receive control when an input/output error occurs on a unit record device. The use of STXIT precludes the use of SYMDMP, STATE, and FLOW in the compiled program and in any other program link-edited with the compiled program, and vice versa.

QUOTE
APOST
: QUOTE indicates to the compiler that the double quotation marks (") should be accepted as the character to delineate literals; APOST indicates that the apostrophe (') should be accepted instead. The compiler will generate the specified character for the figurative constant QUOTE(S).

TRUNC
NOTRUNC
: applies only to COMPUTATIONAL receiving fields in MOVE statements and arithmetic expressions. If TRUNC is specified, extra code is generated to truncate the final intermediate result of the arithmetic expression, or the sending field in the MOVE statement, to the number of digits specified in the PICTURE clause of the COMPUTATIONAL receiving field. If NOTRUNC is specified, the compiler assumes that the data being manipulated conforms to PICTURE and USAGE specifications. The compiler then generates code to manipulate the data based on the size of the field in storage (halfword, etc.). TRUNC conforms to the American National Standard, while NOTRUNC leads to more efficient processing. This will occasionally cause dissimilar results for various sending fields because of the different code generated to perform the operation.

ZWB
NOZWB
: determines if the compiler will generate code to strip the sign when comparing a signed external decimal field to an alphanumeric field. If ZWB is in effect, the signed external

decimal field is moved to an intermediate field and has its sign stripped before being compared to the alphanumeric field. ZWB conforms to the ANS standard, while NOZWB allows the user to test input numeric fields for SPACES to prevent abnormal termination.

SXREF
NOSXREF
causes the compiler to write an alphabetically-ordered cross-reference list on SYSLST. You may want to use the lister cross-reference information in place of this option for large COBOL programs, to decrease run time.

PMAP=h
enables the programmer to request a relocation factor "h". If the PMAP option is specified, the relocation factor is included in the addresses of the object code listing. The relocation factor "h" is a hexadecimal number of from one to eight digits. If the PMAP option is not specified, the relocation factor is assumed to be zero. When PMAP is specified in a segmented program, the listing for segments of priority higher than the segment limit (49, if the SEGMENT-LIMIT clause is not specified), will not be relocated. The PMAP option has meaning only when LISTX or CLIST and/or SYM (for the location of WORKING-STORAGE) is in effect.

OPTIMIZE
NOOPTIMIZE
OPT
NOOPT

OPTIMIZE (OPT) causes optimized object code to be generated by the compiler. The more efficient code generated considerably reduces the amount of space required by the object program. If neither LINK nor DECK is specified in the OPTION statement, then optimized code is not generated by the compiler.

This option cannot be used if either the symbolic debug option (SYMDMP), the statement number option (STATE), or the flow trace option (FLOW[=nn]) is requested.

FLOW[=nn]
provides the programmer with a formatted trace (i.e., a list containing the program identification and statement numbers) corresponding to a variable number of procedures executed prior to an abnormal termination. The value "nn" may range from 0 through 99. If "nn" is not specified, a value of 99 is assumed.

FLOW and STXIT, and FLOW and OPT are mutually exclusive options, i.e., only one may be in effect during a given compilation. In addition, FLOW and STXIT are mutually exclusive at execution time. Additional information on the flow trace option can be found in the chapter "Symbolic Debugging Features."

STATE
NOSTATE
STATE provides the programmer with information about the statement being executed at the time of an abnormal termination of a job. It identifies the program containing the statement and provides the number of the statement and of the verb being executed. STATE and STXIT, STATE and SYMDMP, and STATE and OPT are mutually exclusive options, i.e., no more than one may be in effect during a given compilation. (However, the facilities provided by STATE automatically exist with SYMDMP.) In addition, STATE and STXIT are mutually exclusive at execution time. Additional information on the statement number option can be found in the chapter "Symbolic Debugging Features."

SYNTAX, CSYNTAX, NOSYNTAX,
indicates whether the source text is to be scanned for syntax errors only and appropriate error messages are to be generated. For conditional syntax checking (CSYNTAX), a full compilation is produced so long as no messages exceed the C level. If one or more E-level or higher severity messages are produced, the compiler generates the messages but does not generate object text.

Notes:

1. When the SYNTAX option is in effect, all of the following compile-time options are suppressed:

   OPTION control statement: LINK, DECK, XREF

   CBL statement: SXREF, CLIST, COUNT, VERBREF, VERBSUM

2. When CSYNTAX is requested and one or more D- or E-level messages occur, then the preceding options are suppressed and the CBL option FLAGE is made active.

3. Unconditional syntax checking is assumed if all of the following compile-time options are specified:

OPTION control statement: NOLINK, NOXREF, NODECK

CBL statement: SUPMAP (and CLIST, SXREF, VERBSUM, and VERBREF are not specified)

4. Some compiler diagnostics do not appear when SYNTAX or CSYNTAX is in effect. These are listed in "Program Checkout."

SYMDMP[=filename]
indicates to the compiler that execution-time dumps might be requested for the program currently being compiled. If dumps are desired, the programmer must provide the required control cards at execution time. For storage considerations at execution time, see Appendix F: "System and Size Considerations."

Use of the symbolic debug option necessitates the presence of an additional work file, SYS005, at compile time. The "filename" parameter enables the programmer to specify a name for the SYS005 file that he can retain. If no filename is specified, IJSYS05 will be used. When several COBOL programs are link edited together, the "filename" parameter enables each to have a unique SYMDMP name. Compile and execution must be done in the same job stream. The SYS005 file is deleted at end of job. For a tape file, only unlabeled tapes may be used, and the filename in the SYMDMP=filename parameter is ignored.

SYMDMP and STXIT, SYMDMP and STATE, and SYMDMP and OPT are mutually exclusive options, i.e., no more than one may be in effect during a given compilation. (However, the facilities provided by STATE are automatically included with SYMDMP.) In addition, SYMDMP and STXIT are mutually exclusive at execution-time. Additional information on the symbolic debug option and the required execution-time control cards can be found in the chapter "Symbolic Debugging Features."

Note: If NODECK and NOLINK are requested on the OPTION control statement and either SYMDMP or OPT is specified on the CBL card, the SYMDMP or OPT specification is ignored.

CATALR
NOCATALR
causes the compiler to generate CATALR card images on the SYSPCH file if OPTION DECK is in effect during compilation. This will allow cataloging of the compiler produced object modules into the relocatable library. The module names in the CATALR cards adhere to the same rules as the phase names in the compiler produced PHASE cards according to the segmentation and sort phase naming conventions (see the sections on Sort and Segmentation Features).

LIB
NOLIB
indicates that BASIS and/or COPY statements are in the source program. If either COPY or BASIS is present, LIB must be in effect. If COPY and/or BASIS statements are not present, use of the NOLIB option yields more efficient compiler processing.

VERB
NOVERB
indicates whether procedure-names and verb-names are to be listed with the associated code on the object-program listing. VERB has meaning only if LISTX, CLIST, VERBSUM, VERBREF, COUNT or READY TRACE are in effect. NOVERB yields more efficient compilation.

$\text{LVL=}\left\{\begin{matrix}\text{A}\\\text{B}\\\text{C}\\\text{D}\end{matrix}\right\}$
NOLVL
indicates whether the compiler should identify COBOL clauses and statements in a DOS/VS COBOL source program that do not conform to the Federal Information Processing Standard. FIPS recognizes four language levels: low, low-intermediate, high-intermediate and full. The FIPS Flagger provides four levels of flagging from low (A) to high (D) to conform to the four levels of the FIPS.

Note: The FIPS Flagger needs a disk workfile to be assigned to SYS006.

VERBSUM
NOVERBSUM
provides a brief summary of verbs used in the program and a count of how often each verb was used. This option provides the user with a quick search for specific types of statements. VERBSUM implies VERB.

VERBREF
NOVERBREF
provides a cross reference of all verbs used in the program. This option provides the programmer with a quick index to any verb used in the program. VERBREF implies VERB and VERBSUM.

COUNT
NOCOUNT
generates code to produce verb execution summaries at the end of problem program execution. Each verb is identified by procedure-name and by

statement number, and the number of times it was used is indicated. In addition, the percentage of verb execution for each verb with respect to the execution of all verbs is given. A summary of all executable verbs used in a program and the number of times they are executed is provided. COUNT implies VERB.

Note: If COUNT and STXIT are desired, then either STXIT must be requested in the program unit requesting COUNT, or the program unit requesting COUNT must be entered before the program unit requesting STXIT. See the chapter entitled "Execution Statistics" for additional information on the COUNT option.

LST Statement -- New Compiler Option Card

The LST statement is used to invoke the lister, a portion of the compiler that processes programs written in American National Standard COBOL to produce a reformatted source code listing containing embedded cross-reference information, and uniform indenting conventions.

The LST option card can be placed anywhere between the EXEC statement and the first statement of the COBOL program. It may be placed between any other compiler option cards. The options shown in the following format may appear in any order. Underscoring indicates the default case.

```
┌──────────────────────────────────────────────────────┐
│                                                      │
│[DECK,  ][COPYPCH,  ] [LSTCOMP,] [PROC=1col,]         │
│[NODECK ][NOCOPYPCH ] [LSTONLY ] [      2col]         │
│                                                      │
└──────────────────────────────────────────────────────┘
```

LST
: must begin in column 2 (column 1 must be blank) and be followed by at least one blank.

DECK
NODECK
: indicates whether an updated source deck is to be produced as a result of the lister reformatting and/or the update BASIS library.

COPYPCH
NOCOPYPCH
: will punch updated and reformatted copy libraries as a permanent part of the source when DECK is specified. When no updated source deck is requested, an updated and reformatted COPY library will be punched out.

LSTONLY
LSTCOMP
: when LSTONLY is specified, the program will not be compiled, but a reformatted listing will be produced along with a deck if DECK has been specified. LSTCOMP will provide a source listing and will compile the program as part of the job step. LSTCOMP does not suppress CLIST.

PROC=1col
2col
: will list the Procedure Division in either single- or double-column format. At least 132 print positions are required on the printer for the double-column format.

For more details on the lister program, see the chapter entitled "Using the Lister Feature".

Mutually Exclusive Options

In some of the preceding descriptions of the CBL card options, restrictions have been placed on the use of one option in conjunction with others. It should be noted that if these restrictions are violated, the compiler ignores all but the last of the conflicting options specified. For this reason, if after a CBL card is coded the programmer decides to use a new option that is mutually exclusive with an option on the original CBL card, a new CBL card can be added rather than changing the original card.

Changing the Installation Defaults

In order to change the compiler default options to suit your installation, a new member, C.CBLOPTNS, must be added to the source statement library. This module must contain CBL option cards specifying the desired defaults. Resultant defaults may be overridden at compilation time by supplying a CBL card in the compiler input stream.

Significant Characters for Various Options

The DOS/VS COBOL compiler selects the valid options for processing by looking for three significant characters of each key option word. When the keyword is identified, it is checked for the presence or absence of the prefix NO, as appropriate. The programmer can make the most efficient use of the CBL card by using the significant characters instead of the entire option. Table 3 lists the significant characters for each option.

Table 3. Significant Characters for Various Options

Option	Significant Characters
SEQ	SEQ
FLAGE(W)	LAG,LAGW
BUF	BUF
SPACE	ACE
PMAP	PMA
SUPMAP	SUP
CLIST	CLI
TRUNC	TRU
APOST	APO
QUOTE	QUO
SXREF	SXR
STATE	STA
FLOW	FLO
LIB	LIB
SYMDMP	SYM
OPTIMIZE	OPT
SYNTAX	SYN
CSYNTAX	CSY
VERB	VER
ZWB	ZWB
LVL	LVL
COUNT	COU
VERBSUM	VERBSUM
VERBREF	VERBREF
STXIT	STX
DECK	DEC
COPYPCH	COP
LSTCOMP	STC
LSTONLY	STO
PROC	PRO

Note: SYM on the CBL card should not be confused with SYM on the OPTION card.

JOB CONTROL COMMANDS

Job control commands are distinguished from job control statements by the absence of // blank in positions 1 through 3 of each command. They permit the operator to adjust the system according to day-to-day operating conditions. This is particularly true in the area of device assignment, where the operator may need to (1) communicate to the system that a device is unavailable, or (2) designate a different device as the standard for a given symbolic unit. Therefore, these commands normally are not a part of the regular job deck for a job. Job control commands tend to be effective across jobs, whereas job control statements are confined within a job.

Job control commands are discussed in detail in the publication DOS/VS System Control Statements.

LINKAGE EDITOR CONTROL STATEMENTS

Object modules used as input to the Linkage Editor must include linkage editor control statements. There are four linkage editor control statements: PHASE, INCLUDE, ENTRY, and ACTION.

Linkage editor control statements initially enter the system through the device assigned to SYSRDR as part of the input job stream. PHASE and INCLUDE statements may also be present on SYSIPT or in the relocatable library. All four statements are verified for operation (INCLUDE, ACTION, ENTRY, or PHASE) and are copied to SYSLNK to become input when the Linkage Editor is executed.

Linkage editor control statements must be blank in position 1 of the statement. The operand field is terminated by the first blank position. It cannot extend beyond column 72.

The Linkage Editor is executed as a distinct job step. Figure 5 shows how the linkage editor function is performed as a job step in three kinds of operations.

1. Catalog Programs in Core Image Library. The linkage editor function is performed immediately preceding the operation that catalogs programs into the core image library. When the CATAL option is specified, programs edited by the Linkage Editor are cataloged in the core image library by the Librarian after the editing function is performed. The sequence of this operation is shown in Part A of Figure 5. Note that the input for the LNKEDT function could contain modules from the relocatable library instead of, or in addition to, those modules from the card reader, tape unit, or mass storage unit extent assigned to SYSIPT. This is accomplished by naming the module(s) to be copied from the relocatable library in an INCLUDE statement.

2. Load-and-Execute. The sequence of this operation is shown in Part B of Figure 5. Specifying OPTION LINK causes the Job Control Processor to open SYSLNK, and allows the Job Control Processor to place the object module(s) and linkage editor control statements on SYSLNK. As with the catalog operation, the input can consist of object modules from the relocatable library instead of, or in addition to, those modules from the card reader, tape unit, or disk extent assigned to SYSIPT. This is accomplished by specifying the name of the module to be included in the operand of an INCLUDE statement. After the object modules have been edited and placed in the core image library, the program is executed. The blank operand in the EXEC control statement indicates that the program that has just been link edited and temporarily stored in the core image library is to be executed.

3. Compile-and-Execute. Source modules can be compiled and then executed in a single sequence of job steps. In order to do this, the COBOL compiler is directed to write the object module directly on SYSLNK. This is done by using the LINK option in the OPTION control statement. Upon completion of this output operation, the linkage editor function is performed. The program is link edited and temporarily stored in the core image library. The sequence of this operation is shown in Part C of Figure 5.

In each of the operations described in Figure 5, if a private core image library is assigned, output from the Linkage Editor will be placed (either permanently or temporarily) in the private core image library rather than in the system core image library. If the Linkage Editor is executed in a batched-job foreground partition, a private core image library must be assigned. Private core image libraries are a system generation option.

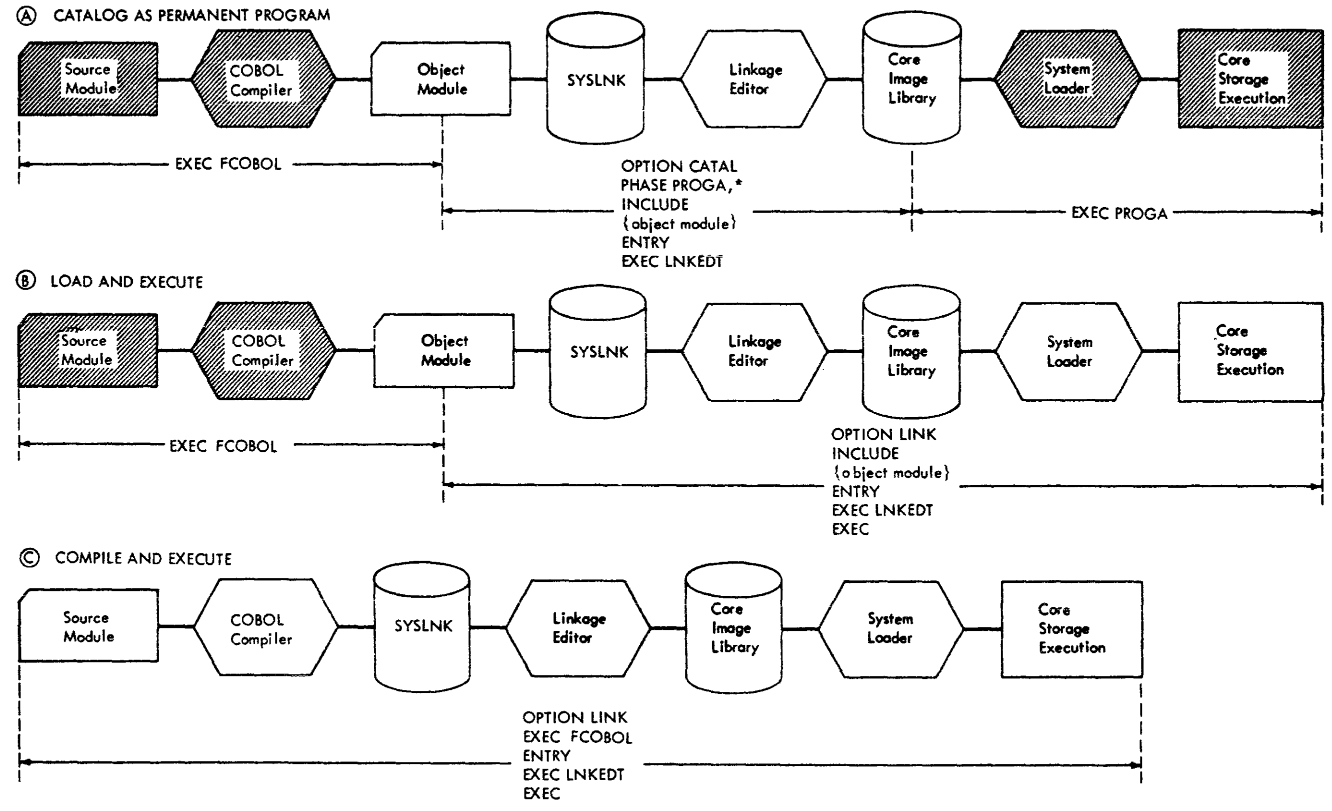


Figure 5. Job Definition -- Use of the Librarian

Control Statement Placement

The placement of linkage editor control statements is subject to the following rules:

1. The ACTION statement must be the first linkage editor control statement encountered in the input stream; otherwise, it is ignored.

2. The PHASE statement must precede each object module that is to begin a phase.

3. The INCLUDE statement must be specified for each object module that is to be included in a program phase.

4. A single ENTRY statement should follow the last object module when multiple object modules are processed in a single linkage editor run.

ACTION and ENTRY statements, when present, must be on SYSRDR. PHASE and INCLUDE statements may be present on SYSRDR, SYSIPT, or in the relocatable library.

PHASE Statement

The PHASE statement must be specified if the output of the Linkage Editor is to consist of more than one phase or if the program phase is to be cataloged in the core image library. Each object module that begins a phase must be preceded by a PHASE statement. Any object module not preceded by a PHASE statement will be included in the current phase.

The statement provides the Linkage Editor with a phase name and an origin point for the phase. The PHASE statement is in the following format:

```
PHASE name,origin[,NOAUTO]
```

name
: is the symbolic name of the phase. It is the name under which the program phase is to be cataloged. This name does not have to be the name specified in the PROGRAM-ID paragraph in the Identification Division of the source program and, in the case of segmentation and/or sort, it should not be the same. It must consist of from one to eight alphanumeric characters. Phases that are to be executed in a segmentation and/or sort structure should have phase names of from five to eight alphanumeric characters, the first four of which should be the same. An asterisk cannot be used as the first character of a phase name. If no phase name is specified, a dummy phase name of PHASE*** is used and execution stops at end of compilation. The job is then cancelled.

origin
: indicates to the Linkage Editor the starting address of this specific phase. An asterisk may be used as an origin specification to indicate that this phase is to follow the previous phase. This origin specification format of the PHASE statement covers all applications that do not include setting up overlay structures. See the chapter "Calling and Called Programs" for information on the PHASE statement for overlay applications.

NOAUTO
: indicates that the Automatic Library Look-Up (AUTOLINK) feature is suppressed for both the private relocatable library and the system relocatable library. (The use of NOAUTO causes the AUTOLINK process to be suppressed for that phase only.) The AUTOLINK feature is discussed later in this chapter.

INCLUDE Statement

The INCLUDE statement must be specified for each object module deck or object module in the relocatable library that is to be included in a program phase. The format of the INCLUDE statement is as follows:

```
INCLUDE [module-name][,(namelist)]
```

The INCLUDE statement has two optional operands. When both operands are used, they must be in the prescribed order. When the first operand is omitted and the second operand is used, a comma must precede the second operand.

module-name
: must be specified when the object module is in the relocatable library. It is not specified when the module to be included is in the form of a card deck being entered from SYSIPT. module-name is the name under which the module was cataloged in the library, and must consist of from one to eight alphanumeric characters.

(namelist)
: causes the Linkage Editor to construct a phase from the control sections specified in the list. Since control sections are of no interest to the COBOL programmer, users interested in this option should refer to the description of the INCLUDE statement in the publication DOS/VS System Control Statements.

ENTRY Statement

The ENTRY statement is required only if the programmer wishes to provide a specific entry point in the first phase produced by the Linkage Editor. When no ENTRY statement is provided, the Job Control Processor writes an ENTRY statement with a blank operand on SYSLNK to ensure that an ENTRY statement will be present to halt link editing. The transfer address will be the load address of the first phase. The ENTRY statement is described further in the publication DOS/VS System Control Statements.

ACTION Statement

The ACTION statement is used to indicate linkage editor options. When used, the statement must be the first linkage editor statement in the input stream. The format of the ACTION statement is as follows:

```
              ⎧ CLEAR  ⎫
              ⎪ MAP    ⎪
              ⎪ NOMAP  ⎪
              ⎪ NOAUTO ⎪
              ⎪ NOREL  ⎪
       ACTION ⎨ CANCEL ⎬
              ⎪ BG     ⎪
              ⎪ F1     ⎪
              ⎪ F2     ⎪
              ⎪ F3     ⎪
              ⎩ F4     ⎭
```

CLEAR
: indicates that the entire temporary portion of the core image library will be set to binary zero before the beginning of the linkage editor function. CLEAR is a time-consuming function and should be used only when necessary.

MAP
: indicates that SYSLST is available for diagnostic messages. In addition, a storage map is output on SYSLST.

NOMAP
: indicates that SYSLST is unavailable when performing the link edit function. The mapping of storage is not performed, and all linkage editor diagnostic messages are listed on SYSLOG.

NOAUTO
: suppresses the AUTOLINK function for both the private and system relocatable libraries during the link editing of the entire program. AUTOLINK is discussed later in this chapter.

CANCEL
: causes an automatic cancellation of the job if any of the linkage editor errors 2100I through 2170I occur. These diagnostic messages can be found in the publication DOS/VS System Control Statements.

BG, F1, F2, F3, and F4
: are options used to link edit a program for execution in a partition other than that in which the link edit function is taking place. See the publication DOS/VS System Control Statements.

NOREL
: suppresses the relocating loader. Link editing for a specific address is performed.

AUTOLINK FEATURE

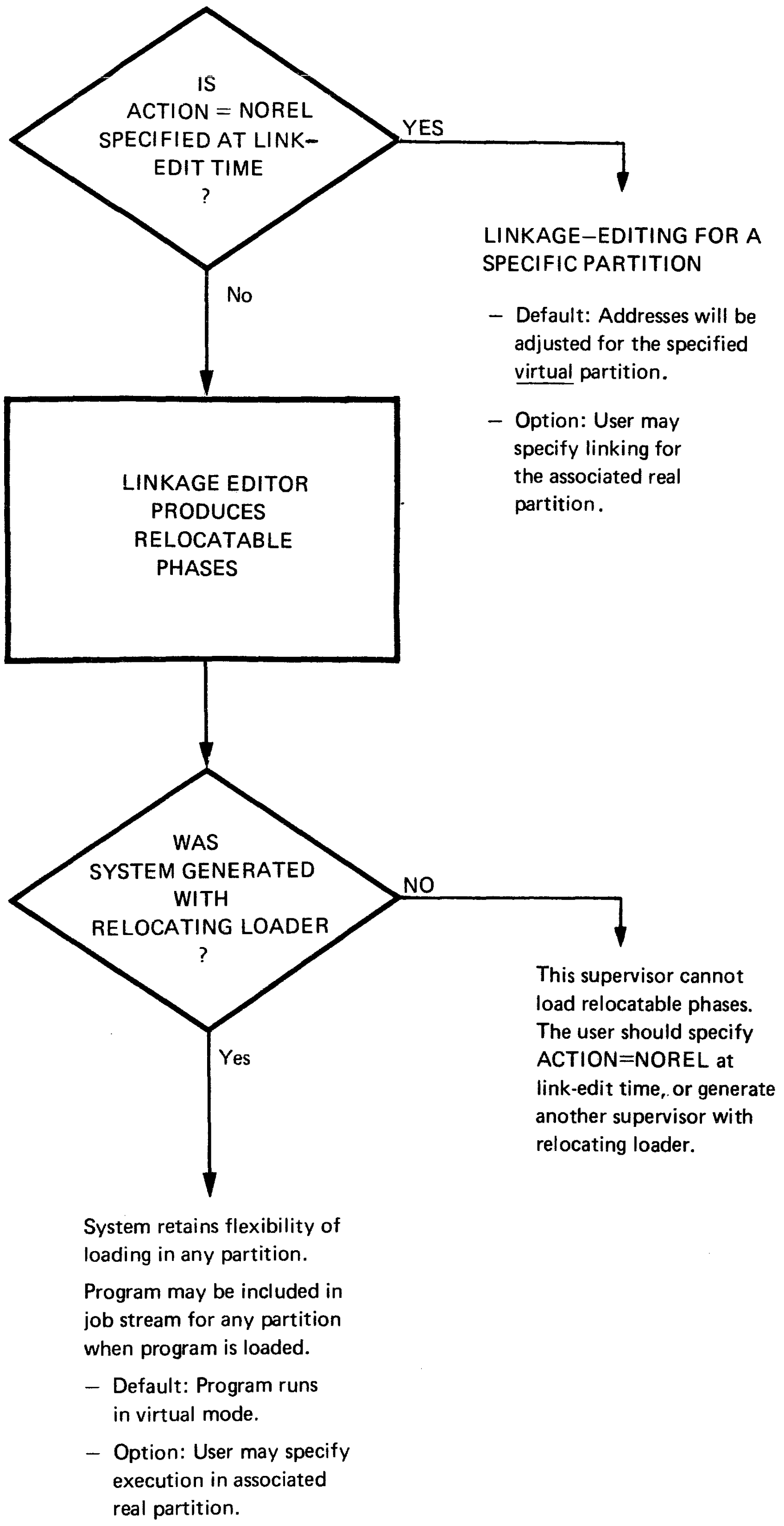
If any references to external-names are still unresolved after all modules have been read from SYSLNK, SYSIPT, and/or the relocatable library, AUTOLINK collects each unresolved external reference from the phase. It then searches the private relocatable library (if SYSRLB has been assigned) and the system relocatable library for module names identical to the unresolved names and includes these modules in the program phase. This feature should not be suppressed (via PHASE or ACTION statements) in linkage editor job steps which include COBOL subroutines cataloged in the relocatable library. See the chapter "Calling and Called Programs" for additional details.

RELOCATING LOADER FEATURE

The relocating loader feature allows users to load single-phase and multi-phase programs at any valid problem program address in the system. Under this option, the linkage editor catalogs relocatable phases into the core image library, and the relocating loader in the supervisor assigns the absolute machine addresses that are necessary for program execution. This means the user need retain only one copy of the program in the core image library.

The relocating loader is an optional feature, and must be specified at system generation time.

Figure 6 illustrates options available during link-editing.


IS ACTION = NOREL SPECIFIED AT LINK-EDIT TIME ?
YES
LINKAGE–EDITING FOR A SPECIFIC PARTITION
– Default: Addresses will be adjusted for the specified virtual partition.
– Option: User may specify linking for the associated real partition.
No
LINKAGE EDITOR PRODUCES RELOCATABLE PHASES
WAS SYSTEM GENERATED WITH RELOCATING LOADER ?
NO
This supervisor cannot load relocatable phases. The user should specify ACTION=NOREL at link-edit time, or generate another supervisor with relocating loader.
Yes
System retains flexibility of loading in any partition.
Program may be included in job stream for any partition when program is loaded.
– Default: Program runs in virtual mode.
– Option: User may specify execution in associated real partition.
Figure 6. Options Available During Link-Editing

DOS/VS supports four libraries: the core image library, the relocatable library, the source statement library, and the procedure library. The core image, relocatable, and source statement libraries are classified as system libraries and private libraries. The procedure library exists only as a system library. The system residence device (SYSRES) contains the system libraries. Private libraries can be contained on separate disk packs. These libraries are discussed under "Private Libraries" in this chapter. Executable programs (core image format) are stored in the core image library; relocatable object modules are stored in the relocatable library; source language routines are stored in the source statement library; catalogued procedures are stored in the procedure library.

PLANNING THE LIBRARIES

The components of the DOS/VS system are shipped in three system libraries: the core image library, the relocatable library, and the source statement library. A fourth library -- the procedure library -- is available but it does not contain any information when the system is shipped. Most programs and procedures developed and used by your installation will also be stored in these libraries. In addition to the system libraries, DOS/VS supports private libraries which you can use to either substitute for or supplement the corresponding system libraries.

Planning the size, contents, and location of these libraries according to the needs of your installation is an essential part of the system generation procedure. Such detailed planning will ensure that:

- No disk space is wasted by components not required in your installation.
- The libraries are large enough to allow for future additions.
- The libraries are accessed by the system with maximum efficiency.

LIBRARIAN

The Librarian is a group of programs that perform three major functions:

1. Maintenance
2. Service
3. Copy

Maintenance functions are used to catalog (that is, add), delete, or rename components of the four libraries, condense libraries and directories, set a condense limit for an automatic condense function, reallocate directory and library extents, and update the source statement and procedure libraries.

The copy function is used either to completely or selectively copy the disk on which the system resides. Service functions are used to translate information from a particular library to printed (displayed) or punched output.

Only the catalog maintenance function of the Librarian is discussed in this publication for the four system libraries. In addition, the update function of the source statement library is discussed. A complete description of librarian functions can be found in the publication DOS/VS System Control Statements.

CORE IMAGE LIBRARY

The core image library may contain any number of programs. Each program consists of one or more separate phases. Associated with the core image library is a core image directory which contains a unique descriptive entry for each phase in the core image library. These entries in the core image directory are used to locate and retrieve phases from the core image library.

Cataloging and Retieving Program Phases -- Core Image Library

If a program is to be cataloged in the core image library, the job control statement // OPTION with the CATAL option must be specified prior to the first linkage editor control card, and must precede the first PHASE card of the program to be cataloged. Upon successful completion of the linkage editor job step, output from the Linkage Editor is placed in the core image library as a permanent member. The program phase is cataloged under the name specified in the PHASE statement.

If a phase in the core image library is to be replaced by a new phase having the same name, only the catalog function need be used. The previously cataloged phase of the same name is implicitly deleted from the core image directory by the catalog function, and the space it occupies in the library can later be released by the condense function.

Note: The necessary ASSGN control statements must follow the // JOB control statement if the current assignments are not the following:

1. SYSRDR -- Card reader, tape unit, or disk extent
2. SYSIPT -- Card reader, tape unit, or disk extent
3. SYSLST -- Printer, tape unit, or disk extent
4. SYSLOG -- Printer keyboard
5. SYSLNK -- Disk extent

The following is an example of cataloging a single phase, FOURA, into the core image library. (The program phase FOURA can be executed in the next job step by specifying the // EXEC statement with a blank name field.)

```
// JOB CATALOG
// OPTION CATAL
   PHASE FOURA,*
   INCLUDE

     {object deck}
/*
// LBLTYP TAPE
// EXEC LNKEDT
// EXEC
/&
```

To compile, link edit, and catalog the phase FOURA into the core image library in the same job, the following job deck could be used:

```
// JOB CATALOG
// OPTION CATAL
   PHASE FOURA,*
// EXEC FCOBOL

     {source deck}
/*
// EXEC LNKEDT
/*
/&
```

When the phase is executed in a subsequent job, the EXEC statement that calls for execution must specify FOURA, i.e., the name by which the phase has been cataloged.

```
// JOB EXJOB
// EXEC FOURA
/&
```

Phases can be in either non-relocatable or relocatable format. The non-relocatable phases are loaded at the address computed at link-edit time into a real or virtual partition. The load addresses and address constants of relocatable phases can be modified by the relocating loader. These phases can be loaded at a virtual address different from the one for which it was link-edited.

RELOCATABLE LIBRARY

The relocatable library contains any number of modules. Each module is a complete object deck in relocatable format. The purpose of the relocatable library is to allow the programmer to maintain frequently used routines in residence and combine them with other modules without recompiling.

Associated with the relocatable library is the relocatable directory. The directory contains a unique, descriptive entry for each module in the relocatable library. The entries in the relocatable directory are used to locate and retrieve modules in the relocatable library.

MAINTENANCE FUNCTIONS

To request a maintenance function for the relocatable library, the following control statement is used:

```
// EXEC MAINT
```

Cataloging a Module -- Relocatable Library

The catalog function adds a module to the relocatable library. A module in the relocatable library is the output of a complete COBOL compilation.

The catalog function implies a delete function. Thus, if a module exists in the relocatable library with the same name as a module to be cataloged, the module in the library is deleted by deleting reference to it in the relocatable directory.

The CATALR control statement is required to add a module to the relocatable library. The format of the CATALR control statement is:

```
    CATALR module-name [,v.m]
```

module-name
: is the name by which the module is known to the control program. The <u>module-name</u> consists of from one to eight characters , the first of which must not be an asterisk.

v.m
: specifies the change level at which the module is to be cataloged. <u>v</u> may be any decimal number from 0 through 127. <u>m</u> may be any decimal number from 0 through 255. If this operand is omitted, a change level of 0.0 is assumed. A change level can be assigned only when a module is cataloged.

All control statements required to catalog an object module must be read from SYSIPT.

<u>Note</u>: If SYSRDR and/or SYSIPT are assigned to a tape unit, the MAINT program assumes that the tape is positioned to the first input record. The tape is not rewound at the end of the job. If a tape mark is found, MAINT assumes end-of-job.

The following is an example of compiling a source program and cataloging the resultant module in the relocatable library. The job deck is read from SYSIPT.

```
// JOB NINE
// OPTION DECK
// EXEC FCOBOL

     {source deck}
/*
// PAUSE PLACE DECK AFTER CATALR CARD
// EXEC MAINT
   CATALR MOD9

     (punched deck goes here)
/*
/&
```

In the above example, as a result of the compile step, the object module is written on SYSPCH. The next job step catalogs the object module (MOD9) into the relocatable library. Since the object module must be cataloged from SYSIPT, a message to the operator instructs him to place the object module on SYSIPT behind the CATALR statement.

The following is an example of cataloging two previously created object modules in the relocatable library:

```
// JOB EIGHT
// EXEC MAINT
   CATALR MOD8A

     {object deck}
   CATALR MOD8B

     {object deck}
/*
/&
```

An additional capability of the system permits a programmer to compile a program and to catalog it to the system relocatable, or private relocatable, library in one continuous run. The programmer inserts a CATALR statement in his job control input stream preceding the compiler execute statement. The CATALR statement will be written on the SYSPCH file (tape or mass storage device) ahead of the compiler output when OPTION DECK is in effect. The programmer then reassigns the SYSPCH file as SYSIPT and executes the MAINT program to perform the catalog function. The output of the compilation (on tape or mass storage device) may be cataloged immediately or it may be cataloged at some later time. It can also be held after cataloging as backup of the compilation.

The preceding method is recommended for single-module object decks. In programs for which the compiler produces multimodule object decks (when segmentation and/or SORT are being used), it is necessary to use the CBL card CATALR option. This option causes a CATALR card to precede each object module.

SOURCE STATEMENT LIBRARY

The source statement library contains any number of books. Each book in the source statement library is composed of a sequence of source language statements. The purpose of the source statement library is to allow the COBOL programmer to initiate the compilation of a book into the source program by using the COPY statement or BASIS card.

Each book in the source statement library is classified as belonging to a specific sublibrary. Sublibraries are defined for three programming languages: Assembler, PL/I, and COBOL. Individual books are classified by sublibrary names. Therefore, books written in each of these languages may have the same name.

Associated with the source statement library is a source statement directory. The directory contains a unique descriptive entry for each book in the source statement library. The entries in the source statement directory are used to locate and retrieve books in the source statement library.

MAINTENANCE FUNCTIONS

To request a maintenance function for the source statement library, the following control statement must be used:

```
// EXEC MAINT
```

Cataloging a Book -- Source Statement Library

The CATALS control statement is required to add a book to a sublibrary of the source statement library.

A book added to a sublibrary of the source statement library is removed by using the delete function. When a book exists in a sublibrary with the same name as a book to be cataloged in that sublibrary, the existing book in the sublibrary is deleted. The following is the format of the CATALS control statement:

```
┌─────────────────────────────────────────┐
│  CATALS sublib.library-name[,v.m[,C]]   │
└─────────────────────────────────────────┘
```

The operation field contains CATALS.

sublib
: represents the sublibrary to which a book is to be cataloged and can be:

  Any alphanumeric character (0-9, A-Z, #, $, and @) representing source statement libraries. The characters A, C, E, and P have special uses:

  A and E are used for the Assembler sublibrary

  C is used for the COBOL sublibrary

  P is used for POWER in PL/I

The sublib qualifier is required. If omitted, the operand will be flagged as invalid and no processing will be done on the book.

library-name
: represents the name of the book to be cataloged. The library-name consists of from one to eight alphanumeric characters, the first of which must be alphabetic. It is the name the programmer uses to retrieve the book when using the source language COPY statement or BASIS card.

v.m
: specifies the change level at which the book is to be cataloged. v may be any decimal number from 0 through 127; m may be any decimal number from 0 through 255. If this operand is omitted, a change level of 0.0 is assumed. The v.m operand becomes part of the entry in the directory for the specified book. Its value is incremented each time an update is performed on the book.

C
: indicates that change level verification is required before updates are accepted for this book.

See the UPDATE control statement, discussed later in this chapter, for its relationship to the v.m and C operands of the CATALS control statement.

In addition to the CATALS control statement, a control statement of the following form must precede and follow the book to be cataloged:

```
BKEND [sublib.library-name],[SEQNCE],
  [count],[CMPRSD]
```

All operand entries are optional. When used, the entries must be in the prescribed order and need appear only in the BKEND statement preceding the book to be cataloged.

The first entry in the operand field is identical to the operand of the CATALS control statement.

SEQNCE
: specifies that columns 76 to 80 of the card images constituting the book are to be checked for ascending sequence numbers. If an error is detected in the sequence checking, an error message is printed. The error can be corrected, and the book can be recataloged.

count
: specifies the number of card images in the book. When the count operand is used, the card input is counted, beginning with preceding BKEND statement and including the subsequent BKEND statement. If an error is detected in the card count, an error message is printed. The error can be corrected, and the book can be recataloged.

CMPRSD
: indicates that the book to be cataloged in the library is in compressed format as a result of CMPRSD having been specified when performing a PUNCH or DSPCH service function. These functions are described in the publication DOS/VS System Control Statements.

Card input for the catalog function is from the device assigned to SYSIPT. The CATALS control statement is also read from the device assigned to SYSIPT.

Frequently used Environment Division, Data Division, and Procedure Division entries can be cataloged in the COBOL sublibrary of the source statement library. A book in the source statement library might consist, for example, of a file description of the Data Division or a paragraph of the Procedure Division.

The following is an example of cataloging a file description in the COBOL sublibrary of the source statement library.

```
// JOB ANYNAME
// EXEC MAINT
   CATALS C.FILEA
   BKEND C.FILEA
            BLOCK CONTAINS 13 RECORDS
            RECORD CONTAINS 120 CHARACTERS
            LABEL RECORDS ARE STANDARD
            DATA RECORD IS RECA.
   BKEND
/*
/&
```

Retrieving a Cataloged Book -- COBOL COPY Statement: The preceding file description can be included in a COBOL source program by writing the following statement:

```
    FD FILEB COPY FILEA.
```

Note that the library entry does not include FD or the file-name. It begins with the first clause that is actually to follow the file-name. This is true for all options of the COPY statement. However, data entries in the library may have a level number (01 or 77) identical to the level number of the data-name that precedes the COPY statement. In this case, all information about the library data-name is copied from the library and all references to the library data-name are replaced by the data-name in the program if the REPLACING option is specified. The change is made only for this program. The entry as it appears in the library remains unchanged. For example, assume the following data entry is cataloged under the library-name DATAR,

```
     01  PAYFILE USAGE IS DISPLAY.
         02  CALC PICTURE 99.
         02  GRADE PICTURE 9
             OCCURS 1 DEPENDING ON CALC OF
             PAYFILE.
           .
```

and the following statement is written in a COBOL source module:

```
     01  GROSS COPY DATAR REPLACING PAYFILE
           BY GROSS.
```

The compiler interprets this as:

```
     01  GROSS USAGE IS DISPLAY.
         02  CALC PICTURE 99.
         02  GRADE PICTURE 9
             OCCURS 1 DEPENDING ON CALC OF
             GROSS.
```

Note also that the library-name is used to identify the book in the library. It has no other use in the COBOL program.

Text cataloged in the source statement library must conform to COBOL margin restrictions.

The COBOL COPY statement is discussed in detail in the section "Extended Source Program Library Facility."

Updating Books -- Source Statement Library

The update function is used to make changes to properly identified statements within a book in the source statement library. Statements are identified in the identification field, columns 73 through 80, which is fixed in format as follows:

Columns 73-76	Program identification which must be constant throughout the book.
Columns 77-80	Sequence number of the statement within the book.

One or more source statements may be added to, deleted from, or replaced in a book in the library without the necessity of replacing the entire book. The update function also provides these facilities:

1. Resequencing statements within a book in the source statement library
2. Changing the change level (v.m) of the book
3. Adding or removing the change level requirement
4. Copying a book with optional retention of the old book with a new name (for backup purposes)

The UPDATE control statement is used for the update function and has the following format:

```
UPDATE sublib.library-name,[s.book1],
   [v.m],[nn]
```

The operation field contains UPDATE.

sublib
: represents the sublibrary that contains the book to be updated. It may be any of the characters 0 through 9, A through Z, #, $, or @.

s.book1
: provides a temporary update option. The old book is renamed s.book1 and the updated book is named sublib.library-name. s indicates the sublibrary that contains the old, renamed book. It may be one of the characters 0 through 9, A through Z, #, $, or @. If this operand is not specified, the old book is deleted.

v.m
: represents the change level of the book to be updated. v may be any decimal number from 0 through 127; m may be any decimal number from 0 through 255. This operand must be present if change level verification is to be performed. Use of the optional entry C in the CATALS control statement at the time the book is cataloged in the library determines whether change level verification is required before updating. If the directory entry specifies that change level verification is not required before updating, the change level operand in the UPDATE control statement is ignored.

If the change level is verified, the change level in the book's directory entry is increased by 1 by the system for verification of the next update. If m is at its maximum value and an update is processed, m is reset to 0 and the value of v is increased by 1. If both v and m are at their maximum values and an update is processed, both v and m are reset to 0.

nn
: represents the resequencing status required for the update. nn may be a 1- or 2-character decimal number from 1 through 10, or it may be the word NO. If nn is a decimal number, it represents the increment that will be used in resequencing the statements in the book. If nn is NO, the statements will not be resequenced. If nn is not specified, the statements will be resequenced with an increment of 1. When a book is resequenced, the sequence number of thefirst statement is 0000. For example, if a book is cataloged in the source statement library with sequence numbers ranging from 0010 through 1000 with increments of 5 for each statement:

and nn is not specified when the update function is performed, the book is resequenced with numbers 0000, 0001, 0002, ... etc.

and NO is specified, insertions, deletions, and/or replacements are made with no effect on the original sequence numbers.

and nn is specified as 2, the book is resequenced with numbers 0000, 0002, 0004, ... etc., regardless of the original sequencing of the book in the library or the sequence numbers of the added or replacement cards.

The UPDATE control statement is followed by ADD, DEL (delete), and/or REP (replace) control statements as required, followed by the terminating END statement. The ADD, DEL, REP, and END statements are identified as update control statements by a right parenthesis in the first position (column 1 in card format). This is a variation from the general librarian control statement format; thus, it clearly identifies these control statements as part of the update function.

ADD Statement: The ADD statement is used for the addition of source statements to a book. The format is:

```
) ADD seq-no
```

ADD indicates that source statements following this statement are to be added to the book.

seq-no
: represents the sequence number of the statement in the book after which the new statements are to be added. It may be any decimal number consisting of from one to four characters.

DEL Statement: The DEL statement causes the deletion of source statements from the book. The format is:

```
) DEL first-seq-no[,last-seq-no]
```

DEL indicates that statements are to be deleted from the book.

first-seq-no
last-seq-no
: represent the sequence numbers of the first and last statements of a section to be deleted. Each number may be a decimal number consisting of from one to four characters. If last-seq-no is not specified, the statement represented by first-seq-no is the only statement deleted.

REP Statement: The REP statement is used when replacement of source statements is required in a book. The format is:

```
) REP first-seq-no[,last-seq-no]
```

REP indicates that source statements following this statement are to replace existing statements in a book.

first-seq-no
last-seq-no
: represent the sequence numbers of the first and last statements of a section to be replaced. Each number may be a decimal number consisting of from one to four characters. Any number of new statements can be added to a book when a section is replaced. (The number of statements added need not equal the number of statements being replaced.)

Sequence number 9999 is the highest number acceptable for a statement to be updated. If the book is so large that statement sequence numbers have "wrapped around" (progressed from 9998, 9999, to 0000,0001), it will not be possible to update statements 0000 and 0001.

END Statement: This statement indicates the end of updates for a given book. The format is:

```
) END [v.m[,C]]
```

v.m
: represents the change level to be assigned to the book after it is updated; v may be any decimal number from 0 through 127. m may be any decimal number from 0 through 255. This operand provides an additional means of specifying the change level of a book in the library. (The other method is through the use of the v.m operand in the CATALS statement.)

C
: indicates that change level verification is required before any subsequent updates for a given book.

If v.m is specified and C is omitted, the book does not require change level verification before a subsequent update. This feature removes a previously specified verification requirement for a particular book.

If both optional operands are omitted, the change level in the book's directory entry is increased as a result of the update, and the verification requirement remains unchanged.

Control Statement Placement: Control statement input for the update function, read from the device assigned to SYSIN, must be in the following order:

1. The JOB control statement.

2. The ASSGN control statements, if the current assignments are not those required. The ASSGN control statements that can be used are SYSIN, SYSLST, and SYSLOG.

3. The EXEC MAINT control statement.

4. The UPDATE control statement.

5.) ADD,) DEL, or) REP statements with appropriate source statements.

6.) END statement.

7. The /* control statement.

8. The /& control statement, which is the last control statement of the job.

The source statement library can also be updated by using the DELETE and INSERT cards. These are discussed in "Extended Source Program Library Facility" in this chapter, and in the publication IBM DOS Full American National Standard COBOL.

UPDATE Function -- Invalid Operand Defaults

UPDATE Statement:

1. If the first or second operand is invalid, the statement is flagged, the book is not updated, and the remaining control statements are checked to determine their validity.

2. If change level verification is required and the incorrect change level is specified, the statement is flagged, the book is not updated, and the remaining control statements are checked to determine their validity.

3. If the resequencing operand is invalid, resequencing is done in increments of 1.

ADD, DEL, or REP Statements:

1. If there is an invalid operation or operand in an ADD, DEL, or REP statement, the statement is flagged, the book is not updated, and the remaining control statements are checked to determine their validity. All options of the UPDATE and END statements are ignored.

2. The second operand must be greater than the first operand in a DEL or REP statement. If not, the statement is considered invalid and is flagged, the book is not updated, and the remaining control statements are checked to determine their validity. All options of the UPDATE and END statements are ignored.

3. All updates to a book between an UPDATE statement and an END statement must be in ascending sequential order of statement sequence numbers. The first operand of a DEL or REP statement must be greater than the last operand of the preceding control statement. The operand of an ADD statement must be equal to or greater than the last operand of the preceding control statement. Consecutive ADD statements must not have the same operand. If these conditions are not met, the default is the same as for items 1 and 2.

END Statement: If the first operand of the END statement is invalid, the statement is flagged, both operands are ignored, and the book is updated as though no operands were specified. If the second operand is invalid, the statement is flagged, the operand is ignored, and the book is updated as though the second operand were not specified.

Out-of-Sequence Updates: If the source statements to be added to a book are not in sequence or do not contain sequence numbers, the book is updated, and a message indicating the error appears following the END statement. If the resequencing option has been specified in the UPDATE statement, the book is sequenced by the specified value, and subsequent updating is possible. If the resequencing option is not specified, the book is resequenced in increments of 1, and subsequent updating will be possible. If the resequencing option NO is specified, the book will be out of sequence, and subsequent updating may not be possible.

The Procedure Library

The procedure library is a new system library that may be used to store -- in card image format --

- Frequently used sets, procedures, of job control and linkage editor statements (basic support).
- Procedures additionally containing inline SYSIPT data, especially control statements for system utility and service programs (extended support). The inline SYSIPT data must be processed under control of the device-independent sequential IOCS or by IBM-supplied service programs and language translators.

The procedure library is part of SYSRES, so the maintenance and service functions available for the other DOS/VS libraries will also support the procedure library.

Cataloged procedures may be included in the job control input stream by a job control statement and temporarily modified by overwrite statements. For more details on cataloged procedures, see DOS/VS System Control Statements.

MAINT, PROCEDURE LIBRARY

To request a maintenance function for the procedure library, use the following EXEC control statement:

```
    // EXEC MAINT
```

One or more of the maintenance functions (catalog, delete, rename, condense, set condense limit, or reallocate) can be requested within a single run. Any number of procedures within the procedure library can be acted upon in this run. Further, one or more of the maintenance functions for either of the other three libraries (core image, source statement, or relocatable) can be requested within this run, for the same MAINT program maintains all four libraries.

Catalog

The control statement required to add a procedure to the procedure library is the CATALP statement. Any number of procedures may be cataloged in a single run. Each procedure must immediately follow the respective CATALP statement.

Statement Format:

```
CATALP procedurename[,VM=v.m][,EOP=yy]
             NO
  ,DATA=YES
```

Each control statement in the procedure library should have a unique identity. This identity is required to modify the job stream at execution time. Therefore, when cataloging, identify each control statement in columns 73-79 (blanks may be embedded).

procedurename
: represents the name of the procedure to be cataloged. The procedurename consists of one to eight alphameric characters, the first of which must be alphabetic. It must not be ALL.

VM=v.m
: specifies the change level at which the procedure is to be cataloged. v may be any decimal number from 0-127. m may be any decimal number from 0-255. If this operand is omitted, a change level of 0.0 is assumed.

  A change level can be assigned only when a procedure is cataloged. The change level is displayed and punched by the service functions.

EOP=yy
: specifies a two-character end-of-procedure delimiter. The EOP parameter can be any combination of characters except /*, /&, //; it must not contain a blank or a comma. The system assumes /+ as default end-of-procedure delimiter. Otherwise you can omit the EOP parameter.

DATA=YES
: specifies that a procedure contains SYSIPT inline data.

  These procedures can only be executed in the extended procedure support.

A procedure to be cataloged into the procedure library may consist of Job Control and linkage editor statements and, if the supervisor was generated with the SYSFIL option, additional control statements for IBM-supplied control and service programs and data processed under control of the device-independent sequential IOCS. The end of a procedure is indicated by the /+ end-of-procedure delimiter or by the end-of-procedure delimiter as specified in the EOP parameter.

If SYSIN is assigned to a tape unit, the MAINT program assumes that the tape is positioned to the first input record. The tape is not rewound at the end of job.

Control statement input for the catalog function, read from the properly assigned device (usually SYSIN), is:

1. the JOB control statement, followed by

2. the ASSGN control statements, if the current assignments are not those required. The ASSGN statements that can be used are SYSIN, SYSLST, and SYSLOG. The ASSGN statements are followed by

3. the EXEC MAINT control statement, followed by

4. the CATALP control statement(s), followed by

5. the module to be cataloged, followed by

6. the /* control statement if other job steps are to follow, or

7. the /& control statement, which is the last control statement of the job.

For example:

```
// JOB CATPROC
      .
      .
      ASSGN control statements,
      if required
      .
      .
// EXEC MAINT
   CATALP PROCA,EOP=AA,DATA=YES
      .
      .
   control statements
      .
      .
   SYSIPT inline data
      .
      .
/* END OF SYSIPT DATA
      .
      .
   control statements
      .
      .
AA END OF PROCEDURE
```

The following restrictions apply when you catalog procedures to the procedure library:

1. A cataloged procedure cannot contain control statements or SYSIPT data for more than one job.

2. If the cataloged control statements include the JOB statement, you must not have a JOB statement when you retrieve the procedure through the EXEC statement. Conversely, if the JOB statement is not cataloged, a JOB statement must precede the EXEC statement that retrieves the procedure.

3. A cataloged procedure must not include any of the following control statements because they are not accepted when the procedure is processed:

```
// ASSGN SYSRDR,X'cuu'
// RESET SYS
// RESET ALL
// RESET SYSRDR
// CLOSE SYSRDR,X'cuu'
┌// ASSGN SYSIPT,X'cuu'┐
│// RESET SYSIPT       │ only if SYSIPT
│                      │ data is
│                      │ included
└// CLOSE SYSIPT,X'cuu'┘
```

4. Cataloged procedures cannot be nested, that is, a cataloged procedure cannot contain an EXEC statement that invokes another cataloged procedure.

Note: Maintenance cannot be performed in the background partition on the procedure library while a foreground partition is using the library.

PSERV, PROCEDURE LIBRARY

To request a service function for the procedure library, use the following EXEC control statement:

```
// EXEC PSERV
```

One or more of the three service functions can be requested within a single run. Any number of procedures within the procedure library can be acted upon in this run.

CALLING CATALOGED PROCEDURES

A cataloged procedure is called by a job that appears in the input stream or via an operator command. The job must consist of a JOB statement and an EXEC statement that specifies the cataloged procedure name. For example:

```
// EXEC  PROC=VCOBCLG
```

The programmer can write cataloged procedures which incorporate job control he used frequently. For example, the programmer may wish to catalog a procedure for compiling, link-editing, and executing a program. It is particularly useful for compiling in a low-priority test partition to which no card reader has been assigned. Using cataloged procedures, the operator can execute via the EXEC statement a cataloged procedure from the console.

PRIVATE LIBRARIES

Private libraries are desirable in the system to permit some libraries to be located on a disk pack other than the one used by SYSRES.

Private libraries are supported for the core image library, the relocatable library, and the source statement library, on the 2311, 2314, 2319, 3330, 3340, fixed block devices, and mass storage devices. However, the following restrictions apply:

1. The private library must be on the same type of disk device as SYSRES; the private core-image library can be on a type of device other than the one SYSRES is on.

2. Reference may be made to a private core image library only if SYSCLB is assigned. If SYSCLB is assigned, the system core image library cannot be changed.

3. Reference may be made to a private relocatable library only if SYSRLB is assigned. If SYSRLB is assigned, the system relocatable library cannot be changed.

4. Reference may be made to a private source statement library only if SYSSLB is assigned. If SYSSLB is assigned, the system source statement library cannot be changed.

5. Private libraries cannot be reallocated.

6. The COPY function is not effective for private libraries except when they are being created.

An unlimited number of private libraries is possible. However, each must be distinguished by a unique file identification in the DLBL statement for the library. No more than one private relocatable library and one private source statement library may be assigned in a given job.

The creation and maintenance of private libraries is discussed in the publication DOS/VS System Control Statements.

Determining the Location of the Libraries

Having decided which libraries you want in your system, you must determine where on the available devices these libraries are to be placed. All system libraries must reside in the SYSRES extent of the system disk pack in a predefined sequence (Figure 7). Although it is theoretically possible to have private libraries on the system pack (outside the SYSRES extent), this is not recommended because it involves increased movement of the disk arm.

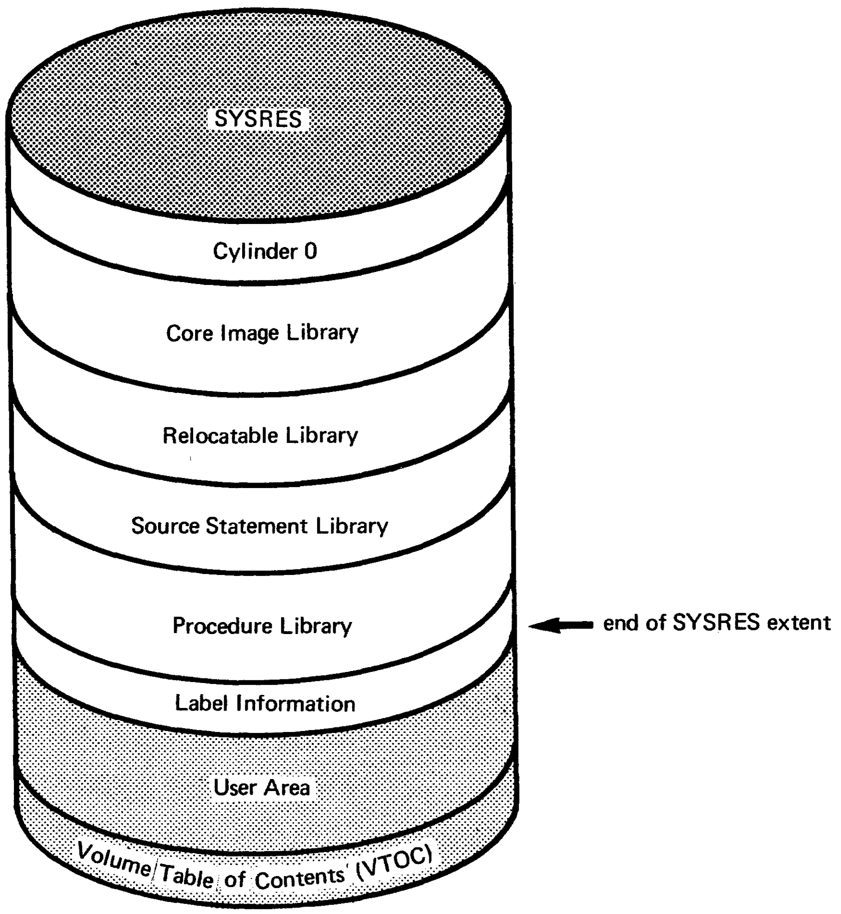


Figure 7. The Relative Location of the Four System Libraries

The directory area for each library is not shown in the Figure 7. By definition, all system libraries reside on the system residence file (SYSRES). If you have additional disk drives, you can define private core image, relocatable, and/or source statement libraries on the extra volumes. These volumes must be of the same type as the SYSRES pack. The system relocatable and system source statement libraries can be removed from SYSRES and established as private libraries; the system core image library, however, must always be present on SYSRES. It can be supplemented but not replaced by a private core image library. The procedure library is supported only as a system library; you cannot create a private procedure library.

SOURCE LANGUAGE CONSIDERATIONS

To use the private source statement library for COPY, BASIS, INSERT, and DELETE(see "Extended Source Program Library Facility" for further details), the ASSGN, DLBL, and EXTENT control statements that define this private library must be present in the job deck for compilation (unless they are permanently set up by the installation). When present, a search for the book is made in the private library. If it is not there, the system library is searched. If the statements for the private library are not present, the system library is searched. A programmer may create several private libraries, but only one private library can be used in a given job.

EXTENDED SOURCE PROGRAM LIBRARY FACILITY

A complete program may be included as an entry in the source statement library by using the catalog function. This program can then be retrieved by a BASIS card and compiled in a subsequent job.

The following control statements would be used to catalog the program SAMPLE as a book in the COBOL sublibrary of the source statement library:

```
// JOB CATALOG
// EXEC MAINT
   CATALS C.SAMPLE
   BKEND C.SAMPLE

      {source program}

   BKEND
/*
/&
```

When compiling a program that has been cataloged in the COBOL sublibrary of the source statement library, a BASIS card brings in an entire source program. The following control statements could be used to compile the cataloged program SAMPLE:

```
// JOB PGM1
// OPTION LOG,DECK,LIST,LISTX,ERRS
// EXEC FCOBOL
 CBL LIB
 BASIS SAMPLE
/*
/&
```

INSERT or DELETE cards may follow the BASIS card if the user wishes to modify the book SAMPLE before it is processed by the compiler. The original source program must have been coded with sequence numbers in columns 1 through 6 of each source card.

The INSERT statement will add new source statements after the specified sequence numbers. The DELETE statement will delete the statements indicated by the sequence numbers, or will delete more than one statement when the first and last sequence numbers to be deleted are specified, separated by a hyphen. Source program cards may follow a DELETE card for insertion before the card following the last one deleted. The sequence numbers in columns 1 through 6 are used to update COBOL source statements at compilation time, and are in effect for the one run only.

Assume that a company runs its payroll program each week as a source program taken from the source statement library. The name of the program is PAYROLL. During a particular year, the old age insurance tax (FICA) is deducted at the rate of 4-2/5% each week for all personnel until earnings exceed $7800. The coding to accomplish this is shown in Figure 8.

Now, however, due to a change in the old age tax laws, tax is to be taken out until earnings exceed $10800 and a new percentage is to be placed. The programmer can code these changes as shown in Figure 9.

The altered program will contain the coding shown in Figure 10.

Reformatted Source Deck

By specifying the DECK option on the LST card, a new COBOL source deck can be produced that reflects the reformatted source listing. This deck may be saved in a BASIS library, used directly as input to the compiler, or punched onto cards. Because of reformatting, the new deck may contain more cards than the original, but the difference is not great enough to cause any appreciable increase in compilation time. The output deck differs from the listing as follows:

1. References, footnotes, and blank lines are omitted.
2. Literals will be repositioned, if needed, to assure proper continuation.
3. Statement numbers are converted to card numbers.
   a. The statement number is multiplied by 10, and leading zeros are added as necessary to fill columns 1 through 6.
   b. Comment and continuation cards are numbered one higher than the preceding card.
   c. Statement-beginning cards are given the higher of the two numbers produced by the first two rules.

The use of this feature avoids having to resequence cards for permanent updating after they have been tested by temporary updating using the BASIS feature; it also avoids the errors incurred during that resequencing process.

```
000730                 IF ANNUAL-PAY GREATER THAN 7800 GO TO PAY-WRITE.
000735                 IF ANNUAL-PAY GREATER THAN 7800 - BASE-PAY GO TO LAST-FICA.
000740   FICA-PAYR.    COMPUTE FICA-PAY = BASE-PAY * .044
000745                       MOVE TAX-PAY TO OUTPUT-TAX.
000750   PAY-WRITE.          MOVE BASE-PAY TO OUTPUT-BASE.
000755                       ADD BASE-PAY TO ANNUAL-PAY.
  .                    .
  .                    .
  .                    .
000850                 STOP RUN.
```

Figure 8. Sample Coding to Calculate FICA

```
// JOB PGM2
// OPTION LOG,DECK,LIST,LISTX,ERRS
// EXEC FCOBOL
 CBL QUOTE, LIB
BASIS PAYROLL
DELETE 000730-000740
000730      IF ANNUAL-PAY GREATER THAN 10800 GO TO PAY-WRITE.
000735      IF ANNUAL-PAY GREATER THAN 10800 - BASE-PAY GO TO LAST-TAX.
000740 TAX-PAYR.  COMPUTE TAX-PAY = BASE-PAY * .0585
/*
```

Figure 9. Altering a Program from the Source Statement Library Using INSERT and DELETE Cards

```
000730              IF ANNUAL-PAY GREATER THAN 10800 GO TO PAY-WRITE.
000735              IF ANNUAL-PAY GREATER THAN 10800 - BASE-PAY GO TO LAST-TAX.
000740 TAX-PAYR.    COMPUTE TAX-PAY = BASE-PAY* .0585.
000750              MOVE TAX-PAY TO OUTPUT-TAX.
000760 PAY-WRITE.   MOVE BASE-PAY TO OUTPUT-BASE.
000770              ADD BASE-PAY TO ANNUAL-PAY.
   .                .
   .                .
   .                .
000850              STOP RUN.
```

Figure 10. Effect of INSERT and DELETE Cards

The DOS/VS COBOL compiler, COBOL object module, Linkage Editor, and other system components can produce output in the form of printed listings, punched card decks, diagnostic or informative messages, and data files directed to tape or to mass storage devices. This chapter gives the format of and describes this output. The same COBOL program is used for each example. "Appendix A: Sample Program Output" shows the output formats in the context of a complete listing generated by the sample program.

COMPILER OUTPUT

The output of the compilation job step may include:

- A printed listing of the job control statements
- A printed listing of the statements contained in the source program
- A glossary of compiler-generated information about data
- Global tables, register assignments, and literal pools
- A printed listing of the object code
- A condensed listing containing only the relative address of the first generated instruction for each verb
- Compiler statistics
- Compiler diagnostic messages
- Cross-reference listings
- System messages
- An object module
- FIPS diagnostic messages

The presence or absence of the above-mentioned types of compiler output is determined by options specified at system generation time. These options can be overridden or additional options specified at compilation time by using the OPTION control statement and the CBL card.

The level of diagnostic message printed depends upon the FLAGW or FLAGE option of the CBL card.

All output to be listed is written on the device assigned to SYSLST. If SYSLST is assigned to a magnetic tape, COBOL will treat the file as an unlabelled tape. Line spacing of the source listing is controlled by the SPACEn option of the CBL card and by SKIP 1/2/3 and EJECT in the COBOL source program. (The lister feature ignores these commands.) The number of lines per page can be specified in the SET command. In addition, a listing of input/output assignments can be printed on SYSLST by using the LISTIO control statement.

On each page of the output, there is a header which contains the PROGRAM-ID, date and time of compilation, as well as an indication of the modification level of the compiler which produced this listing.

Figure 11 contains the compiler output listing shown in "Appendix A: Sample Program Output." Each type of output is numbered, and each format within each type is lettered. The text below and that following the figure is an explanation of the figure.

① <u>The listing of the job control statements associated with this job step</u>. These statements are listed because the LOG option was specified at system generation time.

② <u>Compiler options</u>. The CBL card, if specified, is printed on SYSLST unless the LIST option is suppressed.

③ <u>The source module listing</u>. The statements in the source program are listed exactly as submitted except that a compiler-generated card number is listed to the left of each line. This is the number referenced in diagnostic messages and in the object code listing. It is also the number printed on SYSLST as a result of the source language TRACE statement (if NOVERB is in effect). The source module is not listed when the NOLIST option is specified.

```
// JOB SAMPLE
// OPTION NODECK,LINK,LIST,LISTX,SYM,ERRS   ①
// EXEC FCOBOL


       IBM DOS VS COBOL                          REL 1.0        PP NO. 5746-CB1          07.43.04   03/03/74

 CBL QUOTE,OPT,SXREF,LVL=A  ②
00001    000010 IDENTIFICATION DIVISION.
00002    000020 PROGRAM-ID. TESTRUN.
00003                    AUTHOR.  PROGRAMMER NAME.
00004                    INSTALLATION.  NEW YORK DEVELOPMENT CENTER
00005                    DATE-WRITTEN.  FEBRUARY 18, 1974
00006           DATE-COMPILED. 03/03/74
00007           REMARKS.  THIS PROGRAM HAS BEEN WRITTEN AS A SAMPLE PROGRAM FOR
00008               COBOL USERS.  IT CREATES AN OUTPUT FILE AND READS IT BACK
00009               AS INPUT.
00010    000100
00011    000110 ENVIRONMENT DIVISION.
00012    000120 CONFIGURATION SECTION.
00013    000130 SOURCE-COMPUTER. IBM-370.
00014    000140 OBJECT-COMPUTER. IBM-370.
00015    000150 INPUT-OUTPUT SECTION.
00016    000160 FILE-CONTROL.
00017    000170     SELECT FILE-1 ASSIGN TO SYS008-UT-2400-S.
00018    000180     SELECT FILE-2 ASSIGN TO SYS008-UT-2400-S.
00019    000190
   .
   .
   .                                                                       ③


00056    000550 PROCEDURE DIVISION.
00057           BEGIN.
00058    000570     NOTE THAT THE FOLLOWING OPENS THE OUTPUT FILE TO BE CREATED
00059    000580     AND INITIALIZES COUNTERS.
00060    000590 STEP-1.  OPEN OUTPUT FILE-1.  MOVE ZERO TO KOUNT, NUMBR.
   .
   .
   .


00073    000720 STEP-5. CLOSE FILE-1. OPEN IN.
00074    000730     NOTE THAT THE FOLLOWING READS BACK THE FILE AND SINGLES
00075    000740     OUT EMPLOYEES WITH NO DEPENDENTS.
00076    000750 STEP-6. READ FILE-2 RECORD INTO WORK-RECORD AT END GO TO STEP-8.
00077    000760 STEP-7. IF NO-OF-DEPENDENTS IS EQUAL TO "0" MOVE "Z" TO
00078    000770     NO-OF-DEPENDENTS. EXHIBIT NAMED WORK-RECORD. GO TO STEP-6.
00079    000780 STEP-8. CLOSE FILE-2.
00080    000790     STOP RUN.
```

Figure 11. Examples of Compiler Output (Part 1 of 4)

```
  IBM DOS VS COBOL                          REL 1.0          PP NO. 5746-CB1         07.43.04  10/03/73

     (A)          (B)        (C)                         (D)     (E)       (F)            (G)          (H)              (J)
INTRNL NAME    LVL SOURCE NAME                          BASE    DISPL   INTRNL NAME    DEFINITION     USAGE        R  O  Q  M

DNM=1-148       FD  FILE-1                            DTF=01           DNM=1-148                     DTFMT                   F
DNM=1-179       01  RECORD-1                           BL=1     000     DNM=1-179      DS 0CL20       GROUP
DNM=1-200       02  FIELD-A                            BL=1     000     DNM=1-200      DS 20C         DISP
DNM=1-217       FD  FILE-2                            DTF=02           DNM=1-217                     DTFMT                   F
DNM=1-248       01  RECORD-2                           BL=2     000     DNM=1-248      DS 0CL20       GROUP
DNM=1-269       02  FIELD-A                            BL=2     000     DNM=1-269      DS 20C         DISP
DNM=1-289       01  FILLER                             BL=3     000     DNM=1-289      DS 0CL56       GROUP
DNM=1-308       02  KOUNT                              BL=3     000     DNM=1-308      DS 1H          COMP
DNM=1-323       02  ALPHABET                           BL=3     002     DNM=1-323      DS 26C         DISP
DNM=1-341       02  ALPHA                              BL=3     002     DNM=1-341      DS 1C          DISP         R  O            (4)
DNM=1-359       02  NUMBR                              BL=3     01C     DNM=1-359      DS 1H          COMP
DNM=1-374       02  DEPENDENTS                         BL=3     01E     DNM=1-374      DS 26C         DISP
DNM=1-394       02  DEPEND                             BL=3     01E     DNM=1-394      DS 1C          DISP         R  O
DNM=1-410       01  WORK-RECORD                        BL=3     038     DNM=1-410      DS 0CL20       GROUP
DNM=1-434       02  NAME-FIELD                         BL=3     038     DNM=1-434      DS 1C          DISP
DNM=1-454       02  FILLER                             BL=3     039     DNM=1-454      DS 1C          DISP
DNM=1-473       02  RECORD-NO                          BL=3     03A     DNM=1-473      DS 4C          DISP-NM
DNM=1-492       02  FILLER                             BL=3     03E     DNM=1-492      DS 1C          DISP
DNM=2-000       02  LOCATION                           BL=3     03F     DNM=2-000      DS 3C          DISP
DNM=2-018       02  FILLER                             BL=3     042     DNM=2-018      DS 1C          DISP
DNM=2-037       02  NO-OF-DEPENDENTS                   BL=3     043     DNM=2-037      DS 2C          DISP
DNM=2-063       02  FILLER                             BL=3     045     DNM=2-063      DS 7C          DISP


                    MEMORY MAP
               TGT     (A)             003F8

          SAVE AREA                    003F8
          SWITCH                       00440
          TALLY                        00444
          SORT SAVE                    00448
          ENTRY-SAVE                   0044C
          SORT CORE SIZE               00450
          NSTD-REELS                   00454
          SORT RET                     00456
          WORKING CELLS                00458
          SORT FILE SIZE               00588
          SORT MODE SIZE               0058C
          PGT-VN TBL                   00590
          TGT-VN TBL                   00594
          SORTAB ADDRESS               00598
          LENGTH OF VN TBL             0059C
          LNGTH OF SORTAB              0059E
          PGM ID                       005A0
          A(INIT1)                     005A8
          UPSI SWITCHES                005AC
          DEBUG TABLE PTR              005B4
          CURRENT PRIORITY             005B8
          TA LENGTH                    005B9
          PRBL1 CELL PTR               005BC
          UNUSED                       005C0
          RESERVED                     005C4
          VSAM SAVE AREA ADDRESS       005C8
          UNUSED                       005CC
          RESERVED                     005D4
          OVERFLOW CELLS               005EC
          BL CELLS                     005EC
          DTFADR CELLS                 005F8
          FIB CELLS                    00600
          TEMP STORAGE                 00608
          TEMP STORAGE-2               00610
          TEMP STORAGE-3               00610
          TEMP STORAGE-4               00610                                    (6)
          BLL CELLS                    00610
          VLC CELLS                    00614
          SBL CELLS                    00614
          INDEX CELLS                  00614
          SUBADR CELLS                 00614
          ONCTL CELLS                  0061C
          PFMCTL CELLS                 0061C
          PFMSAV CELLS                 0061C
          VN CELLS                     00620
          SAVE AREA =2                 00624
          XSASW CELLS                  00624


LITERAL POOL (HEX)    (B)

00640 (LIT+0)     00000001  001A5B5B  C2D6D7C5  D5405B5B  C2C3D3D6  E2C55B5B
00658 (LIT+24)    C2C6C3D4  E4D35B5B  C0000000

     DISPLAY LITERALS (BCD)

00664 (LTL+36)    'WORK-RECORD'

               PGT   (C)               00628

          DEBUG LINKAGE AREA           00628
          OVERFLOW CELLS               00628
          VIRTUAL CELLS                0062C
          PROCEDURE NAME CELLS         00638
          GENERATED NAME CELLS         00638
          SUBDTF ADDRESS CELLS         0063C
          VNI CELLS                    0063C
          LITERALS                     00640
          DISPLAY LITERALS             00664
          PROCEDURE BLOCK CELLS        00670
```

Figure 11. Examples of Compiler Output (Part 2 of 4)

```
REGISTER ASSIGNMENT

 REG 6       BL =3      (7)
 REG 7       BL =1
 REG 8       BL =2

WORKING-STORAGE STARTS AT LOCATION 00100 FOR A LENGTH OF 00050.  (5)

 PROCEDURE BLOCK ASSIGNMENT          (8)

  PBL = REG 11

  PBL =1     STARTS AT LOCATION 000674   STATEMENT 60
(A)                 (B)       (C)                          (E)                    (F)
57
            000674                       PN=02    EQU    *
60
            000674                       PN=03    EQU    *
60
            000674                       START    EQU    *
            000674  58 B0 C 048                   L      11,048(0,12)         PBL=1
            000678  58 20 D 1F4                   L      2,1F4(0,13)          BL =1
            00067C  41 10 C 01E                   LA     1,01E(0,12)          LIT+6
            000680  58 00 D 200                   L      0,200(0,13)          DTF=1
            000684  18 40                         LR     4,0
            000686  05 F0                         BALR   15,0
            000688  50 00 F 008                   ST     0,008(0,15)
            00068C  45 00 F 00C                   BAL    0,00C(0,15)
            000690  00000000                      DC     X'00000000'
            000694  0A 02                         SVC    2
            000696  41 00 D 200                   LA     0,200(0,13)          DTF=1
            00069A  58 F0 C 008                   L      15,008(0,12)         V(ILBDIML0)
            00069E  05 EF                         BALR   14,15
            0006A0  58 10 D 200                   L      1,200(0,13)          DTF=1
            0006A4  96 10 1 020                   OI     020(1),X'10'
            0006A8  50 20 D 1F4                   ST     2,1F4(0,13)          BL =1
            0006AC  58 70 D 1F4                   L      7,1F4(0,13)          BL =1
60
            0006B0  D2 01 6 000 C 018             MVC    000(2,6),018(12)     DNM=1-308     LIT+0
            0006B6  D2 01 6 01C C 018  (D)        MVC    01C(2,6),018(12)     DNM=1-359     LIT+0
64
            0006BC                       PN=04    EQU    *
64
            0006BC  48 30 C 01A                   LH     3,01A(0,12)          LIT+2
            0006C0  4A 30 6 000                   AH     3,000(0,6)           DNM=1-308
            0006C4  4E 30 D 210                   CVD    3,210(0,13)          TS=01
            0006C8  D7 05 D 210 D 210             XC     210(6,13),210(13)    TS=01         TS=01
            0006CE  94 0F D 216                   NI     216(13),X'0F'        TS=01+6
            0006D2  4F 30 D 210                   CVB    3,210(0,13)          TS=01
            0006D6  40 30 6 000                   STH    3,000(0,6)           DNM=1-308
            0006DA  48 30 C 01A                   LH     3,01A(0,12)          LIT+2
            0006DE  4A 30 6 01C                   AH     3,01C(0,6)           DNM=1-359
            0006E2  4E 30 D 210                   CVD    3,210(0,13)          TS=01
            0006E6  D7 05 D 210 D 210             XC     210(6,13),210(13)    TS=01         TS=01
            0006EC  94 0F D 216                   NI     216(13),X'0F'        TS=01+6
            0006F0  4F 30 D 210                   CVB    3,210(0,13)          TS=01
            0006F4  40 30 6 01C                   STH    3,01C(0,6)           DNM=1-359
64
            0006F8  41 40 6 002                   LA     4,002(0,6)           DNM=1-341
            0006FC  48 20 6 000                   LH     2,000(0,6)           DNM=1-308
            000700  4C 20 C 01A                   MH     2,01A(0,12)          LIT+2
            000704  1A 42                         AR     4,2
            000706  5B 40 C 018                   S      4,018(0,12)          LIT+0
            00070A  50 40 D 21C                   ST     4,21C(0,13)          SBS=1
            00070E  58 E0 D 21C                   L      14,21C(0,13)         SBS=1
66          000712  D2 00 6 038 E 000             MVC    038(1,6),000(14)     DNM=1-434     DNM=1-341
            000718  41 40 6 01E                   LA     4,01E(0,6)           DNM=1-394
            00071C  48 20 6 000                   LH     2,000(0,6)           DNM=1-308
            000720  4C 20 C 01A                   MH     2,01A(0,12)          LIT+2
            000724  1A 42                         AR     4,2
            000726  5B 40 C 018                   S      4,018(0,12)          LIT+0
            00072A  50 40 D 220                   ST     4,220(0,13)          SBS=2
            00072E  58 F0 D 220                   L      15,220(0,13)         SBS=2
            000732  D2 00 6 043 F 000             MVC    043(1,6),000(15)     DNM=2-37      DNM=1-394
            000738  92 40 6 044                   MVI    044(6),X'40'         DNM=2-37+1
                                                                                         (9)

*STATISTICS*          SOURCE RECORDS =      80      DATA ITEMS =      22      NO OF VERBS =      28
*STATISTICS*          PARTITION SIZE = 655176      LINE COUNT =      56      BUFFER SIZE =     512
*OPTIONS IN EFFECT*   PMAP RELOC ADR =   NONE      SPACING    =       1      FLOW        =    NONE
*OPTIONS IN EFFECT*     LISTX          QUOTE       SYM      NOCATALR       LIST        LINK      NOSTXIT      NOLIB   (10)
*OPTIONS IN EFFECT*    NOCLIST         FLAGW       ZWB      NOSUPMAP       XREF        ERRS       SXREF         OPT
*OPTIONS IN EFFECT*    NOSTATE         TRUNC       SEQ      NOSYMDMP      NODECK      NOVERB    NOSYNTAX      LVL=A
```

Figure 11. Examples of Compiler Output (Part 3 of 4)

```
   IBM DOS VS COBOL                          REL 1.0            PP NO. 5746-CB1         07.43.04  03/03/74

       (A)                                 CROSS-REFERENCE DICTIONARY
  DATA NAMES                      DEFN     REFERENCE

  ALPHA                           000042   000064
  ALPHABET                        000041
  DEPEND                          000045   000066
  DEPENDENTS                      000044
  FIELD-A                         000029
  FIELD-A                         000037
  FILE-1                          000017   000060  000068  000073
  FILE-2                          000018   000073  000076  000079
  KOUNT                           000040   000060  000064  000066  000070
  LOCATION                        000051
  NAME-FIELD                      000047   000064
  NO-OF-DEPENDENTS                000053   000066  000077
  NUMBR                           000043   000060  000064  000067
  RECORD-NO                       000049   000067                              (11)
  RECORD-1                        000028   000068
  RECORD-2                        000036   000076
  WORK-RECORD                     000046   000068  000076  000078


       (B)
  PROCEDURE NAMES                 DEFN     REFERENCE

  BEGIN                           000057
  STEP-1                          000060
  STEP-2                          000064   000070
  STEP-3                          000068   000070
  STEP-4                          000070
  STEP-5                          000073
  STEP-6                          000076   000078
  STEP-7                          000077
  STEP-8                          000079   000076
```

```
      CARD   ERROR MESSAGE (C)            (D)
  (A)    (B)
      00064  ILA5011I-W     HIGH ORDER TRUNCATION MIGHT OCCUR.     (12)
      00064  ILA5011I-W     HIGH ORDER TRUNCATION MIGHT OCCUR.
```

```
                        FEDERAL INFORMATION PROCESSING STANDARDS (FIPS) DIAGNOSTIC MESSAGES              PAGE   1

     LINE   NUMBER     (C)  MESSAGE      (D)
  (A)    (B)
     C0006  ILA8003I-W      DATE-COMPILED PARAGRAPH IS AN EXTENSION TO FIPS LEVEL A.
     00025  ILA8002I-W      RECORDING MODE IS CLAUSE IS AN EXTENSION TO ALL FIPS LEVELS.
     C0034  ILA8002I-W      RECORDING MODE IS CLAUSE IS AN EXTENSION TO ALL FIPS LEVELS.
     C0054  ILA8003I-W      SPACES IS AN EXTENSION TO FIPS LEVEL A.
     C0060  ILA8003I-W      COMMA OR SEMICOLON AS PUNCTUATION IS AN EXTENSION TO FIPS LEVEL A.
     C0062  ILA8003I-W      COMMA OR SEMICOLON AS PUNCTUATION IS AN EXTENSION TO FIPS LEVEL A.
     C0062  ILA8003I-W      COMMA OR SEMICOLON AS PUNCTUATION IS AN EXTENSION TO FIPS LEVEL A.      (13)
     C0064  ILA8003I-W      COMMA OR SEMICOLON AS PUNCTUATION IS AN EXTENSION TO FIPS LEVEL A.
     00064  ILA8003I-W      MULTIPLE RESULTS IN ADD STATEMENT IS AN EXTENSION TO FIPS LEVEL A.
     00068  ILA8003I-W      UPON OPTION OF DISPLAY STATEMENT IS AN EXTENSION TO FIPS LEVEL A.
     C0068  ILA8002I-W      UPON CONSOLE  OPTION OF DISPLAY STATEMENT IS AN EXTENSION TO ALL LEVELS.
     C0068  ILA8003I-W      FROM OPTION OF WRITE STATEMENT IS AN EXTENSION TO FIPS LEVEL A.
     C0070  ILA8003I-W      UNTIL OPTION OF PERFORM STATEMENT IS AN EXTENSION TO FIPS LEVEL A.
     C0076  ILA8003I-W      INTO OPTION OF READ STATEMENT IS AN EXTENSION TO FIPS LEVEL A.
     C0078  ILA8002I-W      EXHIBIT STATEMENT IS AN EXTENSION TO ALL FIPS LEVELS.

     END OF COMPILATION
```

Figure 11. Examples of Compiler Output (Part 4 of 4)

The following notations may appear on the listing:

C	Denotes that the statement was inserted with a COPY statement.
**	Denotes that the card is out of sequence. NOSEQ should be specified on the CBL card if the sequence check is to be suppressed.
I	Denotes that the card was inserted with an INSERT or BASIS card.

If DATE-COMPILED is specified in the Identification Division, any sentences in that paragraph are replaced in the listing by the date of compilation. It is printed in one of the following formats depending upon the format chosen at system generation time.

DATE-COMPILED. month/day/year <u>or</u>

DATE-COMPILED. day/month/year

(4) <u>Glossary</u>. The glossary is listed when the SYM option is specified. The glossary contains information about names in the COBOL source program.

(A) and (F) The internal-name generated by the compiler. This name is used in the compiler object code listing to represent the name used in the source program. It is repeated in column F for readability.

(B) A normalized level number. This level number is determined by the compiler as follows: the first level number of any hierarchy is always 01, and increments for other levels are always by one. Only level numbers 03 through 49 are affected; level numbers 66, 77, and 88, and FD, SD, and RD indicators are not changed.

(C) The data-name that is used in the source module.

<u>Note</u>: The following Report Writer internally-generated data-names can appear under the SOURCE NAME column:

CTL.LVL	Used to coordinate control break activities.
GRP.IND	Used by coding for GROUP INDICATE clause.
TER.COD	Used by coding for TERMINATE clause.
FRS.GEN	Used by coding for GENERATE clause.
-nnnn	Generated report record associated with the file on which the report is to be printed.
RPT.RCD	Build area for print record.
CTL.CHR	First or second position of RPT.RCD. Used for carriage control character.
RPT.LIN	Beginning of actual information which will be displayed. Second or third position of RPT.RCD.
CODE-CELL	Used to hold code specified.
E.nnnn	Name generated from COLUMN clause in 02-level statement.
S.nnnn	Used for elementary level with SUM clause, but not with data-name.
N.nnnn	Used to save the total number of lines used by a report group when relative line numbering is specified.

(D) and (E) For data-names, these columns contain information about the address in the form of a base and displacement. For file-names, the column contains information about the associated DTF or FIB (for VSAM). An indication is also given here if the FD is invalid.

(G) This column defines storage for each data item. It is represented in assembler-like terminology. Table 4 refers to information in this column.

(H) Usage of the data-name. For FD entries, either VSAM is specified, or the DTF type is identified (e.g., DTFDA). For group items containing a USAGE clause, the usage type is printed. For group items that do not contain a USAGE clause, GROUP is printed. For elementary items, the information in the USAGE clause is printed.

Table 4. Glossary Definition and Usage

Type	Definition	Usage
Group Fixed-Length	DS 0CLN	GROUP
Alphabetic	DS NC	DISP
Alphanumeric	DS NC	DISP
Alphanumeric Edited	DS NC	AN-EDIT
Numeric Edited	DS NC	NM-EDIT
Index-Name	DS 1H	INDEX-NM
Group Variable-Length	DS VLI=N	GROUP
Sterling Report	DS NC	RPT-ST
External Decimal	DS NC	DISP-NM
External Floating-Point	DS NC	DISP-FP
Internal Floating-Point	DS 1F	COMP-1
	DS 1D	COMP-2
Binary	DS 1H, 1F, OR 2F	COMP
Internal Decimal	DS NP	COMP-3
Sterling Non-Report	DS NC	DISP-ST
Index-Name	BLANK	INDEX-NAME
File (FD)	BLANK	DTF TYPE
Condition (88)	BLANK	BLANK
Report Definition (RD)	BLANK	BLANK
Sort Definition (SD)	BLANK	BLANK

Note: Under the definition column, N = size in bytes, except in group variable-length where it is a variable cell number.

Ⓙ A letter under column:

R - Indicates that the data-name redefines another data-name.

O - Indicates that an OCCURS clause has been specified for that data-name.

Q - Indicates that the data-name is or contains the DEPENDING ON object of the OCCURS clause.

M - Indicates the record format. This field is not applicable to VSAM. The letters which may appear under column M are:

F - fixed-length records

U - undefined records

V - variable-length records

S - spanned records

⑤ The location and length of WORKING-STORAGE are noted here when CLIST, SYM or LSTX is specified, except under the same conditions as noted below.

⑥ Global tables and literal pool: Global tables and the literal pool are listed when the CLIST, SYM, or LISTX option is specified, unless SUPMAP is specified and an E-level error is encountered, or CSYNTAX is specified and an E-level error is encountered. A global table contains easily addressable information needed by the object program for execution. For example, in the Procedure Division output coding (3), the address of the first instruction under STEP-1 (OPEN OUTPUT FILE-1) is found in the PROCEDURE NAME CELLS portion of the Program Global Table (PGT).

Ⓐ The Task Global Table (TGT). This table is used to record and save information needed during the execution of the object program. This information includes switches, addresses, and work areas.

Ⓑ The Literal Pool. This lists all literals used in the program, with duplications removed. These literals include those specified by the programmer (e.g., MOVE "ABC" TO DATA-NAME) and those generated by the compiler (e.g., to align decimal points in arithmetic computations). The literals are divided into two groups: those that are referenced by instructions (marked "LITERAL POOL") and those that are parameters to the display object time subroutine (marked "DISPLAY LITERALS").

Ⓒ The Program Global Table (PGT). This table contains literals and the addresses of procedure-names, generated procedure-names, and procedure block locators referenced by Procedure Division instructions.

⑦ Register assignment: This lists the permanent register assigned to each base locator in the object program. The remaining base locators are given temporary register assignments but are not listed. Register assignments are listed when CLIST, SYM, or LISTX is specified, and output is not overridden by the same conditions as above.

⑧ Procedure block assignments: Procedure block assignments are printed when OPT is specified. The procedure block assignments give the location within the object program for each block of code addressed by register 11.

⑨ Object code listing. The object code listing is produced when the LISTX option is specified, unless SUPMAP is also specified and an E-level error is encountered, or unless CSYNTAX is specified and an E-level error is encountered. The actual object code listing contains:

Ⓐ The compiler-generated card number. This number identifies the COBOL statement in the source deck which contains the verb that generates the object code found in column C. When VERB is specified, the actual verb or paragraph-name is listed with the generated card number.

Ⓑ The relative location, in hexadecimal notation, of the object code instruction in the module.

Ⓒ The actual object code instruction in hexadecimal notation.

Ⓓ The procedure-name number. A number is assigned only to procedure-names referred to in other Procedure Division statements.

Ⓔ The object code instruction in the form that closely resembles assembler language. (Displacements are in hexadecimal notation.)

Ⓕ Compiler-generated information about the operands of the generated instruction. This includes names and relative locations of literals. Table 5 refers to information in this column.

Table 5. Symbols Used in the Listing and Glossary to Define Compiler-Generated Information

Symbol	Meaning
DNM	SOURCE DATA NAME
SAV	SAVE AREA CELL
SWT	SWITCH CELL
TLY	TALLY CELL
WC	WORKING CELL
TS	TEMPORARY STORAGE CELL
VLC	VARIABLE LENGTH CELL
SBL	SECONDARY BASE LOCATOR
BL	BASE LOCATOR
BLL	BASE LOCATOR FOR LINKAGE SECTION
ON	ON COUNTER
PFM	PERFORM COUNTER
PSV	PERFORM SAVE
VN	VARIABLE PROCEDURE NAME
SBS	SUBSCRIPT ADDRESS
XSW	EXHIBIT SWITCH
XSA	EXHIBIT SAVE AREA
PRM	PARAMETER
PN	SOURCE PROCEDURE NAME
PBL	Procedure Block Locator
GN	GENERATED PROCEDURE NAME
DTF	DTF ADDRESS
FIB	File Information Block (for VSAM)
VNI	VARIABLE NAME INITIALIZATION
LIT	LITERAL
TS2	TEMPORARY STORAGE (NON-ARITHMETIC)
RSV	REPORT SAVE AREA
SDF	Secondary DTF Pointer
TS3	TEMPORARY STORAGE (SYNCHRONIZATION)
TS4	TEMPORARY STORAGE (SYNCHRONIZATION)
INX	INDEX CELL
V(BCDNAME)	ADDRESS CONSTANT
VIR	VIRTUAL
OVF	Overflow Cell

⑩ Statistics: The compiler statistics list the options in effect for this run, the number of Data Division statements specified, and the Procedure Division size. Each level number is counted as one statement in the Data Division. The Procedure Division size is approximately the number of verbs in the Procedure Division.

An indicator is also given here if dictionary spill occurred during compilation. If spill occurred, the amount of storage assigned to the compiler may be increased for better performance. Statistics are not listed if SYNTAX (or CSYNTAX and an E-level or higher error occurred) was in effect.

(11) Cross-reference dictionary: The cross-reference dictionary is produced when the XREF or SXREF option is specified. It is suppressed if CSYNTAX is in effect and an E-level error is encountered. It consists of two parts:

(A) The cross-reference dictionary for data-names consists of data-names followed by the generated card number of the statement which defines each data-name, and the generated card number of statements on which the referenced statement begins. For MOVE CORRESPONDING, the data items actually moved are referenced. Report Writer data-names, with the exception of data-names in the form "-nnn", are defined with the generated card number of their respective RD's.

(B) The cross-reference dictionary for procedure-names consists of the procedure-names followed by the generated card number of the statement where each procedure-name is used as a section-name or paragraph-name, and the generated card number of statements where each procedure-name is referenced.

A reference will appear to a procedure name if there is a reference to a logically equivalent procedure-name; a reference will also appear to a procedure name, if, in a segmented program, an implied branch to a segment entry is made.

If XREF is specified, the names are presented in the order in which they appear in the source program. If SXREF is specified, the names are presented alphabetically. The number of references appearing in the cross-reference dictionary for a given name is based upon the number of times the name is referenced in the code generated by the compiler.

Since a SEARCH verb results in the examination of the individual elements in the named table, the XREF or SXREF for a SEARCH will reference the element name for the table rather than the table itself. LISTER could provide the source cross-reference material that might be desired.

(12) Diagnostic messages: The diagnostic messages associated with the compilation are always listed. The format of the diagnostic message is:

(A) Compiler-generated card number. This is the number of a line in the source program related to the error.

(B) Message identification. The message identification for the DOS/VS COBOL compiler always begins with the symbols ILA.

(C) The severity level. There are four severity levels as follows:

(W) Warning
This level indicates that an error was made in the source program. However, it is not serious enough to interfere with the execution of the program. These warning messages are listed only if the FLAGW option is specified in the CBL card or chosen at system generation time.

(C) Conditional
This level indicates that an error was made but the compiler usually makes a corrective assumption. The statement containing the error is retained. Execution can be attempted.

(E) Error
This level indicates that a serious error was made. Usually the compiler makes no corrective assumption. The statement or option containing the error is dropped. Compilation is completed, but execution of the program should not be attempted.

(D) Disaster
This error indicates that a serious error was made. Compilation is not completed. Results are unpredictable. If this is a compiler error, the job will terminate via the

CANCEL macro and produce a dump.

Ⓓ The message text. The text identifies the condition that caused the error and indicates the action taken by the compiler.

Since Report Writer generates a number of internal data items and procedural statements, some error messages may reflect internal names. In cases where the error occurs mainly in these generated routines, the error messages may indicate the card number of the RD entry for the report under consideration. In addition, there are errors that may indicate the number of the card upon which the statement containing the error ends rather than the card upon which the error occurs. Internal name formats for Report Writer are discussed under "Glossary" (heading 4, item C). Statement numbers are generated when a verb or procedure name is encountered.

The COBOL compile-time message that follows serves as an example of the format of COBOL compiler messages:

```
CARD ERROR MESSAGE

00105 ILA1002I-W ***** SECTION HEADER
                          MISSING.
                    ASSUMED PRESENT.
```

- The code "00105" at the left is the card number of the statement in which the error has occurred. (Some errors may not be discovered until information from various sections of the program is combined. For this reason, the source card number in the error message may not be exact.)
- ILA identifies this as a DOS/VS COBOL compiler message.
- The numeral "1002" represents the identifying number of the message; the first digit of this identifier indicates the phase in which the error was detected. In this case the message was generated by phase 1.
- The symbol "I" means that this is a message to the programmer for his action.
- "W" (warning) is a level of severity in the error codes described in item C.
- The five asterisks (*****) indicate words in a message that vary according to the program being compiled.

The message text is usually composed of two sentences. The first describes the error; the second describes what the compiler has done as a result of the error.

Note: By specifying a PROGRAM-ID of ERRMSG in any source program, the user can generate a complete listing of compiler diagnostics and problem determination aids. (See Figure 12.) In this case, a normal compilation never takes place. Only a list of all error messages and problem determination information is produced. The link option is reset if it was in effect.

Some messages are not given if CSYNTAX or SYNTAX is in effect. See "Program Checkout" for the list of these messages.

⑬ FIPS Diagnostic Messages: The diagnostic messages associated with FIPS are listed separately from the compiler diagnostic messages, with a header identifying them as FIPS diagnostics. The format of the FIPS diagnostic messages is:

Ⓐ Compiler-generated line number. This is the number of a line in the source program containing a nonstandard element.

Ⓑ Message identification. The message identification for FIPS diagnostic messages always begins with the symbols ILA. The identifying numbers of the messages will always be 8001, 8002, 8003, or 8004, where:

1 indicates an extension to a certain level of the FIPS

2 indicates an extension to all levels of the FIPS

3 indicates an extension to one or all levels of the FIPS, or an unusual condition;

4 indicates that there are no FIPS diagnostic messages.

Ⓒ The severity level. All FIPS diagnostic messages have a severity level of W (warning). This level indicates that something in the source program does not conform to the FIPS, but the compilation of the program will not be interrupted.

Ⓓ The message text. The text identifies the condition or element that does not conform to the FIPS. The FIPS level is also designated.

```
// JOB          ERRORMSG  User information
//              EXEC      FCOBOL
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  ERRMSG.
       REMARKS.  COMPILATION OF THIS PROGRAM WILL RESULT IN ALL COMPILER
                 DIAGNOSTICS BEING PRODUCED.  NO OBJECT MODULE AND NO COMPILE-
                 TIME STATISTICS ARE PRODUCED.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       PROCEDURE DIVISION.
      *         THE SAME RESULTS CAN BE ACHIEVED BY CHANGING THE PROGRAM-ID OF
      *         ANY PROGRAM TO 'ERRMSG'.
               STOP RUN.
```

Figure 12. A Program that Produces COBOL Compiler Diagnostics

OBJECT MODULE

The object module contains the external symbol dictionary, the text of the program, and the relocation dictionary. It is followed by an END statement that marks the end of the module. For additional information about the external symbol dictionary and the relocation dictionary, see the publication _DOS/VS System Control Statements_.

An object deck is punched if the DECK option is specified, unless an E-level diagnostic message is generated. The object module is written on SYSLNK if the LINK option is specified, unless an E-level diagnostic message is generated. No deck is punched if CSYNTAX is in effect and E-level errors are encountered, or if SYNTAX is in effect.

LINKAGE EDITOR OUTPUT

The output of the link edit step may include:

- A printed listing of the job control statements
- A map of the phase after it has been processed by the Linkage Editor
- Diagnostic messages
- A listing of the linkage editor control statements
- A phase which may be assigned to the core image library

Any diagnostic messages associated with the Linkage Editor are automatically generated as output. The other forms of output may be requested by the OPTION control statement. All output to be listed is printed on the device assigned to SYSLST.

Figure 13 is an example of a linkage editor output listing. It shows the job control statements and the phase map. The different types of output are numbered and each type to be explained is lettered. The text following the figure is an explanation of the figure.

```
// EXEC LNKEDT ①


JOB  SAMPLE                DOS LINKAGE EDITOR DIAGNOSTIC OF INPUT ②

ACTION TAKEN  MAP REL
LIST   AUTOLINK   IJFFBZZN
LIST   AUTOLINK   ILBDDSP0
LIST   AUTOLINK   IJJCPDV
LIST   AUTOLINK   ILBDDSS0
LIST    INCLUDE IJJCPDV
LIST   AUTOLINK   ILBDIML0
LIST   AUTOLINK   ILBDMNS0
LIST   AUTOLINK   ILBDSAE0
LIST   ENTRY


           Ⓐ       Ⓑ       Ⓒ       Ⓓ       Ⓔ         Ⓕ        Ⓖ          Ⓗ       Ⓙ
         PHASE   XFR-AD  LOCORE  HICORE  DSK-AD     ESD TYPE  LABEL      LOADED  REL-FR

        PHASE***  07D878  07D878  07F1FF  05F OF 4  CSECT     TESTRUN    07D878  07D878  RELOCATABLE

                                                    CSECT     IJFFBZZN   07E1C8  07E1C8
                                                    *  ENTRY  IJFFZZZN   07E1C8
                                                    *  ENTRY  IJFFBZZZ   07E1C8
                                                    *  ENTRY  IJFFZZZZ   07E1C8

                                                    CSECT     ILBDSAE0   07F078  07F078
                                                       ENTRY  ILBDSAE1   07F0C0

                                                    CSECT     ILBDMNS0   07F070  07F070

                                                    CSECT     ILBDIML0   07F018  07F018

                                                    CSECT     ILBDDSP0   07E578  07E578                ③
                                                       ENTRY  ILBDDSP1   07E978

                                                    CSECT     ILBDDSS0   07ECF0  07ECF0
                                                       ENTRY  ILBDDSS1   07EF50
                                                       ENTRY  ILBDDSS2   07EF48
                                                       ENTRY  ILBDDSS3   07F008
                                                       ENTRY  ILBDDSS4   07ED16
                                                       ENTRY  ILBDDSS5   07EDC2
                                                       ENTRY  ILBDDSS6   07EE22
                                                       ENTRY  ILBDDSS7   07EDEC
                                                       ENTRY  ILBDDSS8   07ED46

                                                    CSECT     IJJCPDV    07EAA8  07EAA8
                                                       ENTRY  IJJCPDV1   07EAA8
                                                    *  ENTRY  IJJCPDV2   07EAA8

* UNREFERENCED SYMBOLS                              WXTRN     STXITPSW
            Ⓚ                                       WXTRN     ILBDDBG2
002 UNRESOLVED ADDRESS CONSTANTS
```

Figure 13. Linkage Editor Output

(1) The JOB control statements. These statements are listed because the LOG option is specified.

(2) Disk linkage editor diagnostic message of input. The ACTION statement is not required. If the MAP option is specified, SYSLST must be assigned. If the statement is not used and SYSLST is assigned, MAP is assumed and a storage map and any error diagnostic messages are considered output on SYSLST.

(3) Map of virtual storage. A phase map is printed when MAP is specified (or assumed) during linkage editor processing. The following information is contained in the storage map:

(A) The name of each phase. This is the name specified in the phase statement.

(B) The transfer address of each phase.

(C) The lowest virtual storage location of each phase.

(D) The highest virtual storage location of each phase.

(E) The hexadecimal disk address where the phase begins in the core image library.

(F) The names of all CSECT's belonging to a phase.

(G) All defined entry points within a CSECT. If an entry point is not referenced, it is flagged with an asterisk (*).

(H) The address where each CSECT is loaded.

(J) The relocation factor of each CSECT.

(K) The number of unresolved weak external references. This indication need not concern the programmer. An unresolved weak external reference does not cause the Linkage Editor to use the automatic library call mechanism. Instead, the reference is left unresolved, and the load module is marked as executable. The number of unresolved address constants will not necessarily be the same as the number of unreferenced symbols listed in the Linkage Editor output.

Comments on the Phase Map

The severity of linkage editor diagnostic messages may affect the production of the phase map. Since various processing options affect the structure of the phase, the text of the phase map will sometimes provide additional information. For example, the phase may contain an overlay structure. In this case, a map will be listed for each segment in the overlay structure.

Linkage Editor Messages

The Linkage Editor may generate informative or diagnostic messages. A complete list of these messages is included in the publication DOS/VS System Control Statements.

DOS ANS COBOL Unresolved External References

When the Linkage Editor encounters a weak external reference (WXTRN), autolinking is suppressed and the V-type address constant is either resolved from those modules included into the load module or it remains unresolved. Unresolved WXTRNs will not cause the Linkage Editor to cancel the link step if ACTION CANCEL is in effect.

The DOS/VS COBOL object time subroutine library utilizes WXTRNs not only as address constants but also as switches to determine at object time whether certain options are in effect. It is a very convenient feature which can lead to tight and efficient code.

Unresolved WXTRNs are normally intentional but unresolved EXTRNs are normally unintentional and an error.

Any of the following unresolved WXTRNs may appear when link editing an object module produced by an ANS COBOL compiler:

```
STXITPSW   ILBDFLW2   ILBDMRG0
ILBDDBG2   ILBDSRT0   ILBDFLW3
ILBDADR1   ILBDREL0   ILBDTC00
ILBDDBG0   ILBDTEF0   ILBDTC01
SORTEP     ILBDDSS1   ILBDDBG7
ILBDSTN0   ILBDDSS3   ILBDDBG8
ILBDFLW0   ILBDVOC1   ILBDTC30
```

OUT

COBOL EXECUTION OUTPUT

The output generated by program execution (in addition to data written on output files) may include:

- Data displayed on the console or on the printer
- Diagnostic messages to the programmer
- Messages to the operator
- System informative messages
- SYMDMP, STATE, FLOW, and/or COUNT output
- System diagnostic messages
- A system dump

Appendix I contains the full list of execution time diagnostic messages.

A dump and system diagnostic messages are generated automatically during program execution only if the program contains errors that cause abnormal termination.

SYMDMP output is generated upon request, or upon abnormal termination. STATE and FLOW output are generated upon abnormal termination. The output of these features is discussed in the chapter entitled "Symbolic Debugging Features".

COUNT output is generated upon normal or abnormal termination of the program. Output from this feature is described in the chapter "Execution Statistics".

Figure 14 is an example of output from the execution job step. The following text is an explanation of the illustration.

① <u>Job control statements.</u> These statements are listed because the LOG option is specified.

② <u>Program output on printer</u>. The results of execution of the EXHIBIT NAMED statement appear on the program listing.

③ <u>Console output</u>. Data is printed on the console output unit as a result of the execution of DISPLAY UPON CONSOLE.

OPERATOR MESSAGES

The COBOL phase may issue operator messages. In the message, XX denotes a system-generated 2-character numeric field that is used to identify the program issuing the message.

```
// ASSGN    SYS008,X'483'  } ①
// EXEC

WORK-RECORD = A 0001 NYC Z
WORK-RECORD = B 0002 NYC 1
WORK-RECORD = C 0003 NYC 2
WORK-RECORD = D 0004 NYC 3
WORK-RECORD = E 0005 NYC 4
WORK-RECORD = F 0006 NYC Z
WORK-RECORD = G 0007 NYC 1
WORK-RECORD = H 0008 NYC 2
WORK-RECORD = I 0009 NYC 3
WORK-RECORD = J 0010 NYC 4
WORK-RECORD = K 0011 NYC Z
WORK-RECORD = L 0012 NYC 1
WORK-RECORD = M 0013 NYC 2
WORK-RECORD = N 0014 NYC 3     ②
WORK-RECORD = O 0015 NYC 4
WORK-RECORD = P 0016 NYC Z
WORK-RECORD = Q 0017 NYC 1
WORK-RECORD = R 0018 NYC 2
WORK-RECORD = S 0019 NYC 3
WORK-RECORD = T 0020 NYC 4
WORK-RECORD = U 0021 NYC Z
WORK-RECORD = V 0022 NYC 1
WORK-RECORD = W 0023 NYC 2
WORK-RECORD = X 0024 NYC 3
WORK-RECORD = Y 0025 NYC 4
WORK-RECORD = Z 0026 NYC Z
```

```
BG
BG A 0001 NYC 0
BG B 0002 NYC 1
BG C 0003 NYC 2
BG D 0004 NYC 3
BG E 0005 NYC 4
BG F 0006 NYC 0
BG G 0007 NYC 1
BG H 0008 NYC 2
BG I 0009 NYC 3
BG J 0010 NYC 4
BG K 0011 NYC 0
BG L 0012 NYC 1
BG M 0013 NYC 2
BG N 0014 NYC 3     ③
BG O 0015 NYC 4
BG P 0016 NYC 0
BG Q 0017 NYC 1
BG R 0018 NYC 2
BG S 0019 NYC 3
BG T 0020 NYC 4
BG U 0021 NYC 0
BG V 0022 NYC 1
BG W 0023 NYC 2
BG X 0024 NYC 3
BG Y 0025 NYC 4
BG Z 0026 NYC 0
BG EOJ SAMPLE
   00.56.19,DURATION 00.03.42
```

Figure 14. Output from Execution Job Step

STOP Statement

The following message is generated by the STOP statement with the literal option:

XX C110A STOP 'literal'

Explanation: This message is issued at the programmer's discretion to indicate possible alternative action to be taken by the operator.

Operator Response: Follows the instructions given both by the message and on the job request form supplied by the programmer. If the job is to be resumed, hit the end/enter key.

ACCEPT Statement

The following message is generated by an ACCEPT statement with the FROM CONSOLE option:

XX C111A "AWAITING REPLY"

Explanation: This message is issued by the object program when operator intervention is required.

Operator Response: Enter the reply and hit the end/enter key. (The contents of the text field should be supplied by the programmer on the job request form.) Alphabetic characters may be entered lower case.

SYSTEM OUTPUT

Informative and diagnostic messages may appear in the listing during the execution of the object program.

Each of these messages contains an identification code in the first column of the message to indicate the portion of the operating system that generated the message. Table 6 lists these codes, together with identification for each.

Table 6. System Message Identification Codes

Code	Identification
0	An on-line console message from the Supervisor
1	A message from the Job Control Processor
2	A message from the Linkage Editor
3	A message from the Librarian
4	A message from LIOCS
7	A message from the Sort program
C	A message from COBOL object-time subroutines

OUT

This chapter describes the accepted linkage conventions for calling and called programs and discusses linkage methods when using an assembler language program. In addition, this chapter contains a description of the overlay facility which enables different called programs to occupy the same area in virtual storage at different times. It also contains a suggested assembler language program to be used in conjunction with the overlay feature.

A COBOL source program that passes control to another program is a calling program. The program that receives control from the calling program is referred to as a called program. Both programs must be compiled (or assembled) in separate job steps, but the resulting object modules must be link edited together in the same phase.

A called program can also be a calling program; that is, a called program can, in turn, call another program. In Figure 15 for instance, program A calls program B; program B calls program C. Therefore:

1. A is considered a calling program by B
2. B is considered a called program by A
3. B is considered a calling program by C
4. C is considered a called program by B

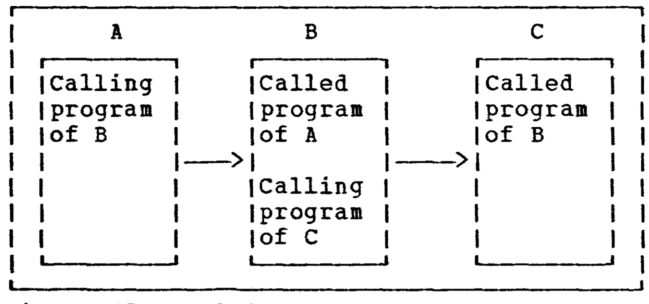


Figure 15. Calling and Called Programs

By convention, a called program may call to an entry point in any other program, except one on a higher level in the "path" of that program. That is, A may call to an entry point in B or C, and B may call C; however, C should not call A or B. Instead, C transfers control only to B by issuing the EXIT PROGRAM or GOBACK statements in COBOL (or its equivalent in another language). B then returns to A.

Compiler-generated switches, e.g., ON and ALTER, are not reinitialized upon each entrance to the called program, that is, the program is in its last executed state.

Note: It is necessary for an American National Standard COBOL program to know whether it is the main or the called program. For this reason, any non-American National Standard COBOL program calling an American National Standard program must first call the subroutine ILBDSET0. The function of this subroutine is to set a switch to X'FF' in subroutine ILBDMNS0, which is the indication to the COBOL program that it is a called program. Standard linkage conventions should be observed when calling ILBDSET0; there are no parameters to be passed.

LINKAGE

Whenever a program calls another program, linkage must be established between the two. The calling program must state the entry point of the called program and must specify any arguments to be passed. The called program must have an entry point and must be able to accept the arguments. Further, the called program must establish the linkage for the return of control to the calling program.

LINKAGE IN A CALLING PROGRAM

A calling COBOL program must contain the following statement at the point where another program is to be called:

```
CALL literal-1 [USING identifier-1
          [identifier-2]...]
```

literal-1
: is the name specified as the program-name in the PROGRAM-ID paragraph of the called program, or the name of the entry point in the called program. When the called program is to be entered at the beginning of the Procedure Division, literal-1 is the name of the program being called. When the called program is to be entered at some point other than the beginning of the Procedure

Division, literal-1 should not be the same as the name specified in the PROGRAM-ID paragraph of the called program. Since the program-name in the PROGRAM-ID paragraph produces an external reference defining an entry point, this entry point name would not be uniquely defined as an external reference.

If the first character of PROGRAM-ID is numeric, the correspondence algorithm is as follows:

```
  0 becomes J
1-9 become A-I
```

Since the system does not include the hyphen as an allowable character, the hyphen is converted to zero if it appears as the second through eighth character of the name.

identifier-1 [identifier-2]...
are the arguments being passed to the called program. Each identifier represents a data item defined in the File, Working-Storage, or Linkage Section of the calling program and should contain a level number 01 or 77. When passing identifiers from the File Section, the file should be open before the CALL statement is executed. If the called program is an assembler language program, the arguments may represent file-names and procedure-names in addition to data-names. If no arguments are to be passed, the USING option is omitted.

LINKAGE IN A CALLED PROGRAM

A called COBOL program must contain two sets of statements:

1. One of the following statements must appear at the point where the program is entered.

   If the called program is entered at the first instruction in the Procedure Division and arguments are passed by the calling program:

```
PROCEDURE DIVISION [USING
  identifier-1 [identifier-2]...].
```

   If the entry point of the called program is not the first statement of the Procedure Division:

```
ENTRY literal-1 [USING identifier-1
  [identifier-2]...]
```

   literal-1
   is the name of the entry point in the called program. It is the same name that appears in the CALL statement of the program that calls this program.

   literal-1 must not be the name of any other entry point or program-name in the run unit.

   identifier-1 [identifier-2]...]
   are the data items representing parameters. They correspond to the arguments of the CALL statement of the calling program. Each data item in this parameter list must be defined in the Linkage Section of the called program and must contain a level number of 01 or 77.

2. Either of the following statements must be inserted where control is to be returned to the calling program:

```
EXIT PROGRAM.
-------------
GOBACK.
```

   Both the EXIT PROGRAM and GOBACK statements cause the restoration of the necessary registers, and return control to the point in the calling program immediately following the calling sequence.

ENTRY POINTS

Each time an entry point is specified in a called program, an external-name is defined. An external-name is a name that can be referenced by another program that has been separately compiled or assembled. Each time an entry name is specified in a calling program, an external reference is defined. An external reference is a symbol that is defined as an external-name in another separately compiled or assembled program. The Linkage Editor resolves external-names and external references, and combines calling and called programs into a format suitable for execution together, i.e., as a single phase.

Note: Several different entry points may be defined in one COBOL source module. Different CALL statements in any module of the phase may specify the same entry point, but each definition of an entry point must be unique in the same phase.

CORRESPONDENCE OF ARGUMENTS AND PARAMETERS

The number of identifiers in the argument list of the calling program should be the same as the number of identifiers in the parameter list of the called program. If the number of identifiers in the argument list of the calling program is greater than the number of identifiers in the parameter list of the called program, only those specified in the parameter list of the called program may be referred to by the called program. There is a one-for-one correspondence. The correspondence is positional and not by name. An identifier must not appear more than once in the same USING clause.

Only the address of an argument is passed. Consequently, both the identifier that is an argument and the identifier that is the corresponding parameter refer to the same location in storage. The pair of identifiers need not be identical, but the data descriptions must be equivalent. For example, if an argument is a level-77 data-name representing a 30-character string, its corresponding parameter could also be a level-77 data-name representing a character string of length 30, or the parameter could be a level-01 data item with subordinate items representing character strings whose combined length is 30.

Although all parameters in the ENTRY statement must be described with level numbers 01 or 77, there is no such restriction made for arguments in the CALL statement. An argument may be a qualified name or a subscripted name. When a group item with a level number other than 01 is specified as an argument, proper boundary word alignment is required if subordinate items are described as COMPUTATIONAL, COMPUTATIONAL-1, or COMPUTATIONAL-2. If the argument corresponds to an 01-level parameter, doubleword alignment is required.

LINK EDITING WITHOUT OVERLAY

Assume that a COBOL main program (COBMAIN), at one or more points in its logic executes CALL statements to COBOL programs SUBPRGA, SUBPRGB, SUBPRGC, and SUBPRGD. Also assume that the module sizes for the main program and subprograms are:

Program	Module Size (in bytes)
COBMAIN	20,000
SUBPRGA	4,000
SUBPRGB	5,000
SUBPRGC	6,000
SUBPRGD	3,000

Through the linkage mechanism, all called programs plus COBMAIN must be link edited together to form one module of 38,000 bytes. Therefore, COBMAIN would require 38,000 bytes of storage in order to be executed. No overlay structure need be specified at link edit time if 38,000 bytes of virtual storage are available.

The following is an example of the job control statements needed to link edit these calling and called programs without specifying an overlay structure. The source decks for COBMAIN and SUBPRGA are included in the job deck, whereas SUBPRGB, SUBPRGC, and SUBPRGD are in the relocatable library.

```
// JOB NOVERLAY
// OPTION LINK,LIST,DUMP
   ACTION MAP
   PHASE EXAMP1,*
   INCLUDE

   {object module COBMAIN}
/*
   INCLUDE SUBPRGB
   INCLUDE SUBPRGC
   INCLUDE SUBPRGD
   INCLUDE

   {object module SUBPRGA}
/*
   ENTRY
// EXEC LNKEDT
// EXEC

   {data for program}
/*
/&
```

Figure 16 is an example of the data flow logic of this call structure where all the programs fit into virtual storage.

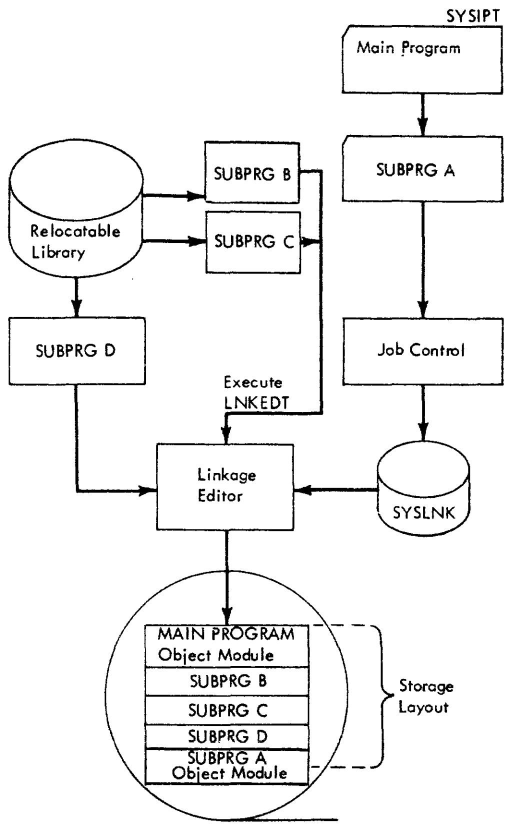


Figure 16. Example of Data Flow Logic in a Call Structure

Note: For the example given, it is assumed that SYSLNK is a standard assignment. The flow diagram illustrates how the various program segments are link edited into storage in a sequential arrangement.

ASSEMBLER LANGUAGE SUBPROGRAMS

A main program written in COBOL can call programs written in other languages that use the same linkage conventions. Whenever a COBOL program calls an assembler language program, certain conventions and techniques must be used.

There are three basic ways to use assembler-written called programs with a main program written in COBOL:

1. A COBOL main program or called program calling an assembler-written program.

2. An assembler-written program calling a COBOL program.

3. An assembler-written program calling another assembler-written program.

From these combinations, more complicated structures can be formed.

In a COBOL program, the expansions of the CALL and GOBACK or EXIT PROGRAM statements provide the save and return coding that is necessary to establish linkage between the calling and called programs in accordance with the linkage conventions of the system. Assembler language programs must be prepared in accordance with the same linkage conventions. These conventions include:

1. Using the proper registers to establish linkage.

2. Reserving, in the calling program, a storage area for items contained in the argument list. This storage area can be referenced by the called program.

3. Reserving, in the calling program, a save area in which the contents of the registers can be saved.

REGISTER USE

The Disk Operating System has assigned functions to certain registers used in linkages. Table 7 shows the conventions for using general registers as linkage registers. The calling program must load the address of the return point into register 14, and it must load the address of the entry point of the called program into register 15.

Table 7. Conventional Use of Linkage Registers

Reg. No.	Reg. Name	Function
1	Argument list register	Address of the argument list passed to the called program.
13	Save area register	Address of the area reserved by the calling program in which the contents of certain registers are stored by the called program.
14	Return register	Address of the location in the calling program to which control is returned after execution of the called program.
15	Entry point register	Address of the entry point in the called program.

SAVE AREA

A calling assembler language program must reserve a save area of 18 words, beginning on a fullword boundary, to be used by the called program for saving registers; it must load the address of this area into register 13. Table 8 shows the layout of the save area and the contents of each word.

A called COBOL program does not save floating-point registers. The programmer is responsible for saving and restoring the contents of these registers in the calling program.

Table 8. Save Area Layout and Word Contents

AREA (word 1)	This word is a part of the standard linkage convention established under the DOS/VS System. The word must be reserved for proper addressing of the subsequent entries. However, an assembler subprogram may use the word for any desired purpose.
AREA+4 (word 2)	The address of the previous save area, that is, the save area of the subprogram that called this one.
AREA+8 (word 3)	The address of the next save area, that is, the save area of the subprogram to which this subprogram refers.
AREA+12 (word 4)	The contents of register 14, that is, the return address.
AREA+16 (word 15)	The contents of register 15, that is, the entry address.
AREA+20 (word 6)	The contents of register 0.
AREA+24 (word 7) . . .	The contents of register 1. . . .
AREA+68 (word 18)	The contents of register 12.

ARGUMENT LIST

The argument list is a group of contiguous fullwords, beginning on a fullword boundary, each of which is an address of a data item to be passed to the called program. If the program is to pass arguments, an argument list must be prepared and its address loaded into register 1. The high-order bit of the last argument, by convention, is set to 1 to indicate the end of the list.

Any assembler-written program must be coded with a detailed knowledge of the data formats of the arguments being passed. Most coding errors occur because of the data format discrepancies of the arguments.

If one programmer writes both the calling program and the called program, the data format of the arguments should not present a problem when passed as parameters. However, when the programs are written by different programmers, the data format specifications for the arguments must be clearly defined for the programmer.

The linkage conventions used by an assembler program that calls another program are illustrated in Figure 16. The linkage should include:

1. The calling sequence.
2. The save and return routines.
3. The out-of-line parameter list. (An in-line parameter list may be used.)
4. A save area on a fullword boundary.

FILE-NAME AND PROCEDURE-NAME ARGUMENTS

A calling COBOL program that calls an assembler-language program can pass file-names and procedure-names, in addition to data-names, as identifiers. In the actual identifier-list that the compiler generates, the procedure-name is passed as the address of the procedure. For a file, the address of the DTF is passed, and the user must ensure that the file is already open. A VSAM file-name may not be passed.

Care must be taken when using these options. The user must be thoroughly familiar with the generated coding for each option and statement, as well as the structure of the object program.

```
deckname  START   0                 INITIATES PROGRAM ASSEMBLAGE AT FIRST
*                                   AVAILABLE LOCATION.  ENTRY POINT TO THE
*                                   PROGRAM.
          ENTRY   name1
          EXTRN   name2
          USING   name1,15
*  SAVE ROUTINE
name1     STM     14,r1,12(13)      THE CONTENTS OF REGISTERS 14, 15, AND
*                                   0 THROUGH r1 ARE STORED IN THE SAVE
*                                   AREA OF THE CALLING PROGRAM (PREVIOUS
*                                   SAVE AREA). r1 IS ANY NUMBER FROM 0 THROUGH 12.
          LR      r3,15
          DROP    15
          USING   name1,r3          WHERE r3 AND r2 HAVE BEEN SAVED
          LR      r2,13             LOADS REGISTER 13, WHICH POINTS TO THE
*                                   SAVE AREA OF THE CALLING PROGRAM, INTO
*                                   ANY GENERAL REGISTER, r2, EXCEPT 0 AND 13.
          LA      13,AREA           LOADS THE ADDRESS OF THIS PROGRAM'S
*                                   SAVE AREA INTO REGISTER 13.
          ST      13,8(r2)          STORES THE ADDRESS OF THIS PROGRAM'S SAVE
*                                   AREA INTO WORD 3 OF THE SAVE AREA OF THE
*                                   CALLING PROGRAM.
          ST      r2,4(13)          STORES THE ADDRESS OF THE PREVIOUS SAVE
*                                   AREA (I.E., THE SAME AREA OF THE CALLING
*                                   PROGRAM) INTO WORD 2 OF THIS PROGRAM'S
*                                   SAVE AREA.
          BC      15,prob1
AREA      DS      18F               RESERVES 18 WORDS FOR THE SAVE AREA
*                                   THIS IS LAST STATEMENT OF SAVE ROUTINE.
prob1     {User-written program statements}
          L       15,VCON           INDICATE COBOL PROGRAM IS
          BALR    14,15             A SUBPROGRAM
*  CALLING SEQUENCE
          LA      1,ARGLST
          L       15,ADCON
          BALR    14,15
          {Remainder of user-written program statements}
*  RETURN ROUTINE
          L       13,4(13)          LOADS THE ADDRESS OF THE PREVIOUS SAVE
*                                   AREA BACK INTO REGISTER 13.
          LM      2,r1,28(13)       THE CONTENTS OF REGISTER 2 THROUGH r1 ARE
*                                   RESTORED FROM THE PREVIOUS SAVE AREA.
          L       14,12(13)         LOADS THE RETURN ADDRESS, WHICH IS IN
*                                   WORD 4 OF THE CALLING PROGRAM'S SAVE AREA,
*                                   INTO REGISTER 14.
          MVI     12(13),X'FF'      SETS FLAG FF IN THE SAVE AREA OF THE
*                                   CALLING PROGRAM TO INDICATE THAT CONTROL
*                                   HAS RETURNED TO THE CALLING PROGRAM.
          BCR     15,14             LAST STATEMENT IN RETURN ROUTINE
VCON      DC      V(ILBDSET0)
ADCON     DC      A(name2)          CONTAINS THE ADDRESS OF SUBPROGRAM name2.
*  PARAMETER LIST
ARGLST    DC      AL4(arg1)         FIRST STATEMENT IN PARAMETER AREA SETUP
          DC      AL4(arg2)
           .
           .
           .
          DC      X'80'             FIRST BYTE OF LAST ARGUMENT SETS BIT 0 TO 1
          DC      AL3(argn)         LAST STATEMENT IN PARAMETER AREA SETUP
```

Figure 17. Sample Linkage Routines Used with a Calling Subprogram

```
ADCON     DC      A(prob1)
           .
           .
          LA      14,RETURN
          L       15,ADCON
          CNOP    2,4
          BALR    1,15
          DC      AL4(arg1)
          DC      AL4(arg2)
           .
           .
          DC      X'80'
          DC      AL3(argn)
RETURN    EQU     *
```

Figure 18. Sample In-line Parameter List

In-Line Parameter List

The assembler programmer may establish an in-line parameter list instead of an out-of-line list. In this case, he may substitute the calling sequence and parameter list illustrated in Figure 18 for that shown in Figure 17.

LOWEST LEVEL PROGRAM

If an assembler called program does not call any other program (i.e., if it is at the lowest level), the programmer should omit the save routine, calling sequence, and parameter list shown in Figure 17. If the assembler called program uses any registers, it must save them. Figure 19 illustrates the appropriate linkage conventions used by an assembler program at the lowest level.

```
deckname  START   0
          ENTRY   name

          USING   *,15
name      STM     14,r1,12(13)
           .
           .
User-written program statements
           .
           .
          LM      2,r1,28(13)
          MVI     12(13),X'FF'
          BCR     15,14
```

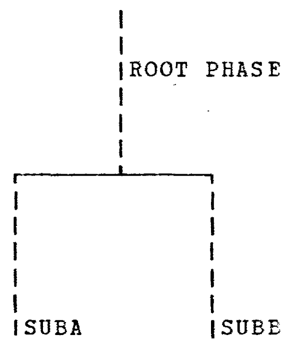
Note: If registers 13 and/or 14 are used in the called subprogram, their contents should be saved and restored by the called subprogram.

Figure 19. Sample Linkage Routines Used with a Lowest Level Subprogram

OVERLAYS

If a program is too large to be contained in the number of bytes available in virtual ~~storage~~, it can still be executed by means of an overlay structure. An overlay structure permits the re-use of storage locations previously occupied by another program. In order to use an overlay structure, the programmer must plan his program so that one or more called programs need not be in storage at the same time as the rest of the program phase. The programmer should reassess, when going to VS, whether programs which used to require an overlay structure still do.

The following is a diagram of the basic form of a program to be overlaid:



The root phase consists of the COBOL main program and an assembler language subroutine which handles the overlay structures. SUBA and SUBB are the called programs that are to be overlaid in storage.

In using the overlay technique, the programmer specifies to the Linkage Editor which programs are to overlay each other. These programs are processed by the Linkage Editor so they can be placed automatically in storage for execution when called by the main program. The resulting output of the Linkage Editor is called an overlay structure.

SPECIAL CONSIDERATIONS WHEN USING OVERLAY STRUCTURES

There are three areas of special concern to the programmer who decides to use the overlay feature. These problems concern the use of the assembler language subroutine, proper link editing, and job control statements.

ASSEMBLER LANGUAGE SUBROUTINE FOR ACCOMPLISHING OVERLAY

The CALL statement is used for "direct" linkage; that is, the assistance of the Supervisor is not required (as it is when loading or fetching a phase). There are no COBOL statements that will generate the equivalent of the LOAD or FETCH assembler macro instructions. For this reason, one must call an assembler program to effect an overlay of a COBOL program. This routine must be link edited as part of either a root phase or permanently resident phase.

The sample overlay subroutine shown in Figure 20 is governed by the following restrictions:

1. The example is a suggested technique, and is not the only technique.
2. It can be used for assembler overlays if the programmer has a desired entry point in his END card and the first statement at that entry point is 'STM 14,12,12(13)'.
3. This subroutine can be used for a COBOL program which contains an ENTRY statement immediately following the Procedure Division header. It will not work with a COBOL subprogram compiled with a Procedure Division USING statement or for entry points in a COBOL subprogram which appear anywhere other than as the first instruction of the Procedure Division. A suggested technique for diverse entry points is a table look-up using V-type constants.

```
STMNT       SOURCE STATEMENT

0001  OVERLAY   START 0
0002            ENTRY OVRLAY
0003  * AT ENTRY TIME
0004  *     R1=POINTER TO ADCON LIST OF USING ARGUMENTS
0005  *     FIRST ARGUMENT IS PHASE OR SUBROUTINE NAME
0006  *     MUST BE 8 BYTES
0007  *     R13=ADDRESS OF SAVE AREA
0008  *     R14=RETURN POINT OF CALLING PROGRAM
0009  *     R15=ENTRY POINT OF OVERLAY PROGRAM
0010  * AT EXIT
0011  *     R1=POINTER TO SECOND ARGUMENT OF ADCON LIST
0012  *         OF USING ARGUMENTS
0013  *     R14=RETURN POINT OF CALLING PROGRAM--NOT THIS PROG
0014  *     R15=ENTRY POINT OF PHASE OR SUBPROGRAM
0015  *
0016            USING *,15
0017  OVRLAY    STM  0,1,SAVE           SAVE WORK REGS
0018            L    1,0(1)             POINT R1 TO PHASE NAME
0019            CLC  CORSUB,0(1)        IN CORE?
0020            BE   SUBIN              YES,BR
0021            MVC  CORSUB(8),0(1)     SET CURRENT PHASE
0022            SR   0,0
0023            SVC  4                  LOAD PHASE
0024  SEARCH1   LA   1,4(1)             STEP SEARCH POINT
0025            CLC  0(3,1),=C'COB'     END OF INIT1?
0026            BNE  SEARCH1            NO, LOOP
0027            S    1,=F'8'            POINT TO "START" ADCON
0028            L    1,0(1)             LOAD "START"
0029            LA   1,8(1)             INCREMENT TO "ENTRY"
0030            ST   1,ASUB             SAVE ENTRY ADDRESS
0031  SUBIN     LM   0,1,SAVE           RELOAD WORK REGS
0032            LA   1,4(1)             POINT TO PARAMETERS
0033            L    15,ASUB
0034            BR   15                 BRANCH TO ENTRY POINT
0035  CORSUB    DS   0CL8
0036            DC   8X'FF'
0037  ASUB      DS   F
0038  SAVE      DS   2F
0039            END
```

Figure 20. Example of an Assembler Language Subroutine for Accomplishing Overlay

Note: Care should be taken with the techniques used in statements 0019 and 0020. Only when the COBOL program is loaded are altered GO TO statements reinitialized. A better technique would be to load the called programs each time they are required.

The examples given in Figures 20, 21, and 22 require that all overlay modules be linked together. To permit linkage to and return from modules, compiled and link edited separately, the following changes to Figure 20 are necessary:

Replace lines 25 through 28

```
    CLC  COBCON,20(1)      END OF INIT?
    BNE  SEARCH1           NO, LOOP
    LR   0,1               SAVE ADDR ADCON INIT1
    L    1,0(1)            GET INIT1 ADDR
    MVC  NOP+3(1),139(1)   GET DISP OF VIRT CELL
    LR   1,0               RESTORE ADDR OF ADCON INIT1
    L    1,4(1)            GET ADDR OF PGT
NOP L    1,0(1)            LOAD ADDR OF ILBDMNS0
    MVI  0(1),X'FF'        SET 'CALLED PROGR' FLAG
    LR   1,0               RESTORE ADDR OF ADCON INIT1
    L    1,12(1)           LOAD 'START' ADDRESS
```

Insert after line 38

```
COBCON DC              CL3'COB'
```

LINK EDITING WITH OVERLAY

In a linkage editor job step, the programmer specifies the overlay points in a program by using PHASE statements. In the Working-Storage Section, a level-01 or level-77 constant must be created for each phase to be called at execution time. These constants have a PICTURE of X(8) and a VALUE clause containing the same name as that appearing on the PHASE card for that segment in the link edit run.

In addition, each argument to be passed to the called program must have an entry in the Linkage Section. Remember, also, that the ENTRY statement should not refer to the program-name. (Use of the program-name will result in incorrect execution.)

When more than one subprogram in the overlay structure requires the same COBOL subroutine, the // EXEC LNKEDT statement must be preceded by INCLUDE cards for each of these subroutines. The names of these subroutines can be determined by requesting LISTX at compile time.

When preparing the control cards for the Linkage Editor, the programmer should be certain to include the assembler language subroutine with the main (root) phase. Also, to achieve maximum overlay, the phase names for the called programs should be different from the names of the called programs specified in the PROGRAM-ID paragraphs.

Figure 21 is a flow diagram of the overlay logic. The PHASE cards indicate the beginning address of each phase. The phases OVERLAYC and OVERLAYD will have the same beginning address as OVERLAYB. The sequence of events is:

1. The main program calls the overlay routine.

2. The overlay routine fetches the particular COBOL subprogram and places it in the overlay area.

3. The overlay routine transfers control to the first instruction of the called program.

4. The called program returns to the COBOL calling program (not to the assembler language overlay routine).

If OVERLAYB were known to be in storage, the CALL statement would be:

```
CALL "OVERLAYB" USING PARAM-1, PARAM-2.
```

But when using the assembler language overlay routine (OVRLAY), it becomes:

```
   CALL "OVRLAY" USING PROCESS-LABEL,
      PARM-1, PARM-2.
```

where PROCESS-LABEL contains the external-name OVERLAYB of the called program.

However, the ENTRY statement of the called program is the same for both cases, i.e., ENTRY "OVERLAYB" USING PARAM-1, PARAM-2, whether it is called indirectly by the main program through the overlay program or called directly by the main program.

Note: An ENTRY which is to be called by OVRLAY must precede the first executable statement in the called program.

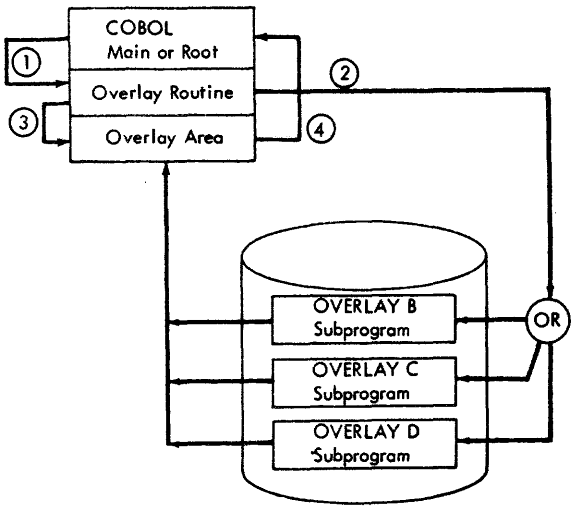


Figure 21. Flow Diagram of Overlay Logic

JOB CONTROL FOR ACCOMPLISHING OVERLAY

The job control statements required to accomplish the overlay illustrated in Figure 21 are shown in Figure 22. The PHASE statements specify to the Linkage Editor that the overlay structure to be established is one in which the called programs OVERLAYB, OVERLAYC, and OVERLAYD overlay each other when called during execution.

Note: The phase name specified in the PHASE card must be the same as the value contained in the first argument for CALL "OVRLAY", i.e., PROCESS-LABEL, COMPUTE-TAX, etc., contain OVERLAYB, OVERLAYC, respectively, which are the names given in the PHASE card.

It is the programmer's responsibility to write the entire overlay, i.e., the COBOL main (or calling) program and an assembler language subroutine (for which a sample program is given in this chapter) that fetches and overlays the called programs. A calling sequence to obtain an overlay structure between three COBOL subprograms is illustrated in Figure 23.

```
// JOB OVERLAYS
// OPTION LINK
   PHASE OVERLAY,ROOT
// EXEC FCOBOL
    {COBOL Source for Main Program MAINLINE}
/*
// EXEC ASSEMBLY
     [Source deck for Assembler Language Routine OVERLAY]
/*
   PHASE OVERLAYB,*
// EXEC FCOBOL
     {COBOL Source for Called Program OVERLAYB}
/*
   PHASE OVERLAYC,OVERLAYB
// EXEC FCOBOL
     {COBOL Source for Called Program OVERLAYC}
/*
   PHASE OVERLAYD,OVERLAYC
// EXEC FCOBOL
     {COBOL Source for Called Program OVERLAYD}
/*
// EXEC LNKEDT
// EXEC
/*
/&
```

Figure 22. Job Control for Accomplishing Overlay

```
COBOL Program Main (Root or Main Program)

IDENTIFICATION DIVISION.
PROGRAM-ID. MAINLINE.
.
.
.
ENVIRONMENT DIVISION.
.
.
.
DATA DIVISION.
.
.
.
WORKING-STORAGE SECTION.
77  PROCESS-LABEL PICTURE IS X(8) VALUE IS "OVERLAYB".
77  PARAM-1 PICTURE IS X.
77  PARAM-2 PICTURE IS XX.
77  COMPUTE-TAX PICTURE IS X(8) VALUE IS "OVERLAYC".

01  NAMET.
    02 EMPLY-NUMB PICTURE IS 9(5).
    02 SALARY PICTURE IS 9(4)V99.
    02 RATE PICTURE IS 9(3)V99.
    02 HOURS-REG PICTURE IS 9(3)V99.
    02 HOURS-OT PICTURE IS 9(2)V99.
01  COMPUTE-SALARY PICTURE IS X(8) VALUE IS "OVERLAYD".
01  NAMES.
    02 RATES PICTURE IS 9(6).
    02 HOURS PICTURE IS 9(3)V99.
    02 SALARYX PICTURE IS 9(2)V99.
.
.
.
PROCEDURE DIVISION.
.
.
.
    CALL "OVRLAY" USING PROCESS-LABEL, PARAM-1, PARAM-2.
.
.
.
    CALL "OVRLAY" USING COMPUTE-TAX, NAMET.
.
.
.
    CALL "OVRLAY" USING COMPUTE-SALARY, NAMES.
.
.
.
```

Figure 23. Calling Sequence to Obtain Overlay Between Three COBOL Subprograms (Part 1 of 3)

```
COBOL Subprogram B

IDENTIFICATION DIVISION.
PROGRAM-ID. OVERLAY1.
.
.
.
ENVIRONMENT DIVISION.
.
.
.
DATA DIVISION.
.
.
LINKAGE SECTION.

01  PARAM-10 PICTURE X.
01  PARAM-20 PICTURE XX.
.
.
.
PROCEDURE DIVISION.
PARA-NAME.  ENTRY "OVERLAYB" USING PARAM-10, PARAM-20.
.
.
.
             GOBACK.


COBOL Subprogram C

IDENTIFICATION DIVISION.
PROGRAM-ID. OVERLAY2.
.
.
.
 ENVIRONMENT DIVISION.
.
.
.
.
DATA DIVISION.
.
.
LINKAGE SECTION.

01  NAMEX.
    02 EMPLY-NUMBX PICTURE IS 9(5).
    02 SALARYX PICTURE IS 9(4) V99.
    02 RATEX PICTURE IS 9(3)V99.
    02 HOURS-REGX   PICTURE IS 9(3)V99.
    02 HOURS-OTX PICTURE IS 9(2)V99.

PROCEDURE DIVISION.
PARA-NAME. ENTRY "OVERLAYC" USING NAMEX.
.
.
.
             GOBACK.
```

Figure 23. Calling Sequence to Obtain Overlay Between Three COBOL Subprograms (Part 2 of 3)

```
COBOL Subprogram D

IDENTIFICATION DIVISION.
PROGRAM-ID. OVERLAY3.
.
.
.
ENVIRONMENT DIVISION.
.
.
.
DATA DIVISION.
.
.
LINKAGE SECTION
01  NAMES.
    02 RATES PICTURE IS 9(6).
    02 HOURS PICTURE IS 9(3)V99.
    02 SALARYX PICTURE IS 9(2)V99.
.
PROCEDURE DIVISION.
PARA-NAME. ENTRY "OVERLAYD" USING NAMES.
.
.
.
           GOBACK.
```

Figure 23. Calling Sequence to Obtain Overlay Between Three COBOL Subprograms (Part 3 of 3)

COBOL segmentation permits the user to subdivide logically and physically the Procedure Division of a COBOL object program. All source sections which contain the same segment-number in their section headers will be considered at object time to be one segment. Since segment-numbers can range from 00 through 99, it is possible to subdivide any object program into a maximum of 100 segments.

Program segments may be of three types: fixed permanent, fixed overlayable, and independent as determined by the programmer's assignment of segment numbers.

Segmentation of a program would be used when virtual storage is limited. In a real storage system, the following would apply:

1. Fixed segments are always in real storage during the execution of the entire program, that is, they cannot be overlayed except when the system itself is executing another program, in which case fixed segments may be "rolled out."

2. Fixed overlayable segments may be overlayed during program execution, but any such overlaying is transparent to the user, that is, they are logically identical to fixed segments, but physically different from them.

3. Independent segments may be overlayed, but such overlaying will result in the initialization of that segment. Therefore, independent segments are logically different from fixed permanent/fixed overlayable segments, and physically different from fixed segments.

In a virtual storage system, all logically "fixed" segments, that is, fixed permanent and fixed overlayable, are treated the same. They are both "paged in and out" as required for execution.

In the same manner, independent segments are paged in and out; when they are paged in, however, they are brought back in the initial state.

In DOS/VS COBOL, segments that are overlayed are not actually "paged out". All the variable data items associated with the segment are contained in one segment, which is considered the root segment. When a segment is "paged in", all the fields which must be reinitialized are contained in the root segment. Thus no fields in other than the root segment are modified.

The program SAVECORE could be segmented as illustrated in Figure 24.

```
IDENTIFICATION DIVISION.

PROGRAM-ID. SAVECORE.
.
.
ENVIRONMENT DIVISION.

OBJECT-COMPUTER. IBM-370.
     SEGMENT-LIMIT IS 15.
.
.
DATA DIVISION.
.
.
PROCEDURE DIVISION.
SECTION-1  SECTION 8.
.
.
SECTION-2  SECTION 8.
.
.
SECTION-3  SECTION 16.
.
.
SECTION-4  SECTION 8.
.
.
SECTION-5  SECTION 50.
.
.
SECTION-6  SECTION 16.
.
.
SECTION-7  SECTION 50.
.
.
```

Figure 24. Segmenting the Program SAVECORE

Assuming that 12K of virtual storage is available for the program SAVECORE, Figure 25 shows the manner in which storage would be utilized. It is apparent from the illustration that SECTION-3, SECTION-6, and SECTION-7 cannot be in storage at the same time, nor can SECTION-3, SECTION-5 and SECTION-7 be in storage simultaneously.

Sections in the permanent segment (SECTION-1, SECTION-2, and SECTION-4) are those which must be available for reference at all times, or which are referenced frequently. They are distinguished here by the fact that they have been assigned priority numbers less than the segment limit.

Sections in the overlayable fixed segment are sections which are less frequently used. They are always made available in the state they were in when last used. They are distinguishable here by the fact that they have been assigned priority numbers greater than the segment limit but less than 49.

Sections in the independent segment can overlay, and be overlaid by, either an overlayable fixed segment or another independent segment. Independent segments are those assigned priority numbers greater than 49 and less than 100, and they are always given control in their initial state.

OPERATION

Execution of the object program begins in the root segment. The first segment in the permanent segment is considered the root segment. If the program does not contain a permanent segment, the compiler generates a dummy segment which will initiate the execution of the first overlayable or independent segment. All global tables, literals, and data areas are part of the root segment. Called object time subroutines are also part of the root segment. When CALL statements appear in a segmented program, subprograms are loaded with the fixed portion of the main program as if they had a priority of zero.

Segmented programs must not be called by another program (segmented or not segmented). If a segmented program calls a subprogram, the CALL statement may appear in any segment. However, the object module associated with the subprogram must be included in the root segment prior to the execution of the main program. This can be accomplished in either of two ways as follows:

1. Produce object decks for both programs and place the one for the subprogram in the root segment:

   ```
   PHASE,ROOT
   ESD card for the root segment

   {object deck for the main program}

   {object deck for the subprogram}

   followed by a // EXEC LNKEDT and a //
      EXEC.
   ```

2. Catalog the object module for the subprogram in the relocatable library prior to link editing the main program. Insert an INCLUDE card for the subprogram and an ENTRY card for the root phase into the linkage editor control cards for the root phase of the main program. The ENTRY card will cause the linkage editor to pass control to the main program at execution time. The Linkage Editor will search the relocatable library for the subprogram and include it with the root phase.

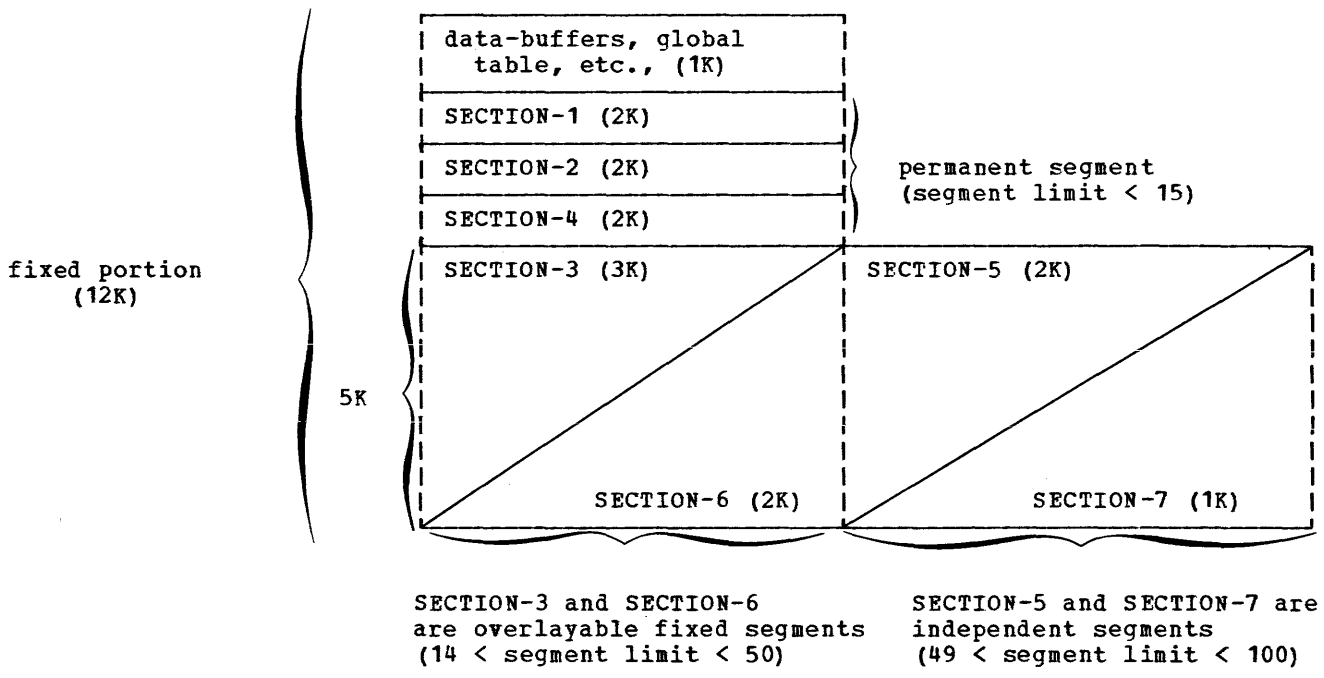

data-buffers, global table, etc., (1K)
SECTION-1 (2K)
SECTION-2 (2K)
SECTION-4 (2K)
permanent segment (segment limit < 15)
fixed portion (12K)
5K
SECTION-3 (3K)
SECTION-5 (2K)
SECTION-6 (2K)
SECTION-7 (1K)
SECTION-3 and SECTION-6 are overlayable fixed segments (14 < segment limit < 50)
SECTION-5 and SECTION-7 are independent segments (49 < segment limit < 100)

Figure 25. Storage Layout for SAVECORE

OUTPUT FROM A SEGMENTED PROGRAM

COMPILER OUTPUT

The output produced by the compiler is an overlay structure consisting of multiple object modules preceded by linkage editor control statements. Segments whose priority is greater than the segment limit (or 49, if no SEGMENT-LIMIT clause is specified) consist of executable instructions only.

The compiler generates each segment as a separate object module preceded by a PHASE card. The names appearing on these PHASE cards (segment-names) conform to the following naming conventions:

1. The name of the root segment is the same as the program-name specified in the PROGRAM-ID clause.

2. The name of each overlayable and independent segment is a combination of the program-name and the priority number of the segment. These names are formed according to the following rules:

   a. If the program-name is 6, 7, or 8 characters in length, the segment-name consists of the first 6 characters of program-name plus the 2-character priority number.

   b. If the program-name is less than 6 characters in length, the priority number is appended after the program-name.

   c. Since the system expects the first character of PROGRAM-ID to be alphabetic, the first character, if numeric, is converted as follows:

      0 -> J
      1-9 -> A-I

      The hyphen is converted to zero if it appears as the second through eighth character.

   d. When DECK is specified, the punched object deck is sequenced according to segments. Columns 73-74 contain the first two characters of the program-id, columns 75-76 contain the priority number of the segment, and columns 77-80 contain the sequence number of the card. The priority of the root segment is punched as 00.

e. When the compiler option CATALR is in effect, the PHASE card for each segment is preceded by a CATALR card with the same name. This will enable direct cataloging of the compiler-produced object module into the relocatable library from which a load module may be link edited into the core-image library.

Note: Single-digit priority numbers are preceded by a zero.

Warning: In order to avoid duplicate names, the programmer must be aware of the above naming conventions. If the last two characters of an 8-character PROGRAM-ID are numeric, these same two characters may not appear in the source program as a segment number.

Figure 26 is an illustration of the compiler output for the skeleton program shown in Figure 24.

```
PHASE SAVECORE,ROOT

 {object module for the root segment
  (sections with priority-numbers less
  than the segment limit) including any
  programs called by SAVECORE}


PHASE SAVECO16,*

 {object module for segments with a
  priority of 16 (two sections)}


PHASE SAVECO50,SAVECO16

 {object module for segments with a
  priority of 50 (two sections)}
```

Figure 26. Compiler Output for SAVECORE

LINKAGE EDITOR OUTPUT

Figure 27 is an illustration of the input to the Linkage Editor and the phase map produced by the Linkage Editor resulting from the compilation and editing of the segmented program BIGJOB. The following text is an explanation of the figure.

(1) PHASE card generated by the compiler for the root segment BIGJOB.

(2) AUTOLINK card for the Segmentation subroutine.

(3) PHASE cards generated by the compiler for segments of priority 10, 47-50, 60, 62, and 63.

(4) Control card generated for the Sort Feature. This card is explained in "Sort in a Segmented Program."

(5) Location of the entry point CURSEGM. Item 5 is explained in "Determining the Priority of the Last Segment Loaded into the Transient Area."

(6) Load address of phase BIGJOB00. Item 6 is explained in "Sort in a Segmented Program."

Note: If the CATALR option of the CBL card is specified, the compiler generates CATALR cards in front of PHASE cards.

Cataloging a Segmented Program

When the CATAL option is used to catalog a segmented program, the following points should be observed:

1. To avoid duplicate names, the programmer must be aware of the naming conventions used by the compiler (see "Compiler Output") because a segment-name may be the same as a phase-name already existing in the core image library.

2. Since the PHASE card is generated by the compiler, the programmer must not specify a PHASE card for the program.

To invoke a previously cataloged segmented program, the programmer must use the following control statement:

```
// EXEC name
```

where name is the program-name specified in the PROGRAM-ID clause.

Determining the Priority of the Last Segment Loaded into the Transient Area

If a segmented program is abnormally terminated during execution, and the SYMDMP option has been specified, the CURRENT PRIORITY cell in the Task Global Table contains the priority of the last segment loaded into the transient area. If SYMDMP has not been specified, the priority of this segment can be determined as follows:

1. In the map of virtual storage generated by the Linkage Editor, under the column LABEL, look for the name 'CURSEGM' (see item 5 in Figure 27).

2. Associated with this label, in the column LOADED, is an address.

3. At this location is stored the priority (one byte) of the segment current in the transient area. If this byte is X'00', no segment has been loaded into the transient area. This indicates that the error causing the dump occurred in the root segment.

SORT IN A SEGMENTED PROGRAM

If a segmented program contains a SORT statement, the sort program will be loaded above the largest overlayable or independent segment as shown in Figure 28.

The compiler accomplishes this by providing the following control statement at the end of the overlay structure:

```
   PHASE BIGJOB00,transient area + L
```

This card is illustrated in Figure 27, item 4. The value of "L" in the figure is X'002F2' which is the length of the longest segment, BIGJOB47, rounded to the next halfword boundary. Note that Linkage Editor relocates the phase BIGJOB00 to the next doubleword boundary (see Figure 27, item 6).

Using the PERFORM Statement in a Segmented Program

When the PERFORM statement is used in a segmented program, the programmer should be aware of the following:

- A PERFORM statement that appears in a section whose priority-number is less than the segment limit can have within its range only (a) sections with priority-numbers less than 50, and (b) sections wholly contained in a single segment whose priority-number is greater than 49.

  Note: As an extension to American National Standard COBOL, DOS/VS COBOL allows sections with any priority-number to fall within the range of a PERFORM statement.

- A PERFORM statement that appears in a section whose priority-number is equal to or greater than the segment limit can have within its range only (a) sections with the same priority-number as the section containing the PERFORM statement, and (b) sections with priority-numbers that are less than the segment limit.

  Note: As an extension to American National Standard COBOL, DOS/VS COBOL allows sections with any priority-number to fall within the range of a PERFORM statement.

- When a procedure-name in a permanent segment (priority-number less than segment limit) is referred to by a PERFORM statement in an independent segment (priority-number greater than 49), the independent segment is reinitialized upon exit from the PERFORM. When a PERFORM statement in the overlayable-fixed segment (priority-number greater than segment limit and less than 50) refers to a procedure-name in a permanent segment, the overlayable-fixed segment is not reinitialized upon exit from the PERFORM.

```
JOB  BIGJ                     DISK LINKAGE EDITOR DIAGNOSTIC OF INPUT

ACTION TAKEN       MAP

LIST    PHASE  BIGJOB,ROOT<---(1)
.
.
.
LIST    AUTOLINK      ILBDSEM0<---(2)
LIST    AUTOLINK      ILBDSRT0
.
.
LIST    PHASE BIGJOB10,*
LIST    PHASE BIGJOB47,BIGJOB10 \
LIST    PHASE BIGJOB48,BIGJOB47 |
LIST    PHASE BIGJOB49,BIGJOB48 |
LIST    PHASE BIGJOB50,BIGJOB49 ><---(3)
LIST    PHASE BIGJOB60,BIGJOB50 |
LIST    PHASE BIGJOB62,BIGJOB60 |
LIST    PHASE BIGJOB63,BIGJOB62 /
LIST    PHASE BIGJOB00,BIGJOB63+X'002F2'<---(4)

        PHASE     XFR-AD  LOCORE  HICORE  DSK-AD    ESD TYPE  LABEL     LOADED  REL-FR

ROOT  BIGJOB      003000  003000  0075A3  64 04 1   CSECT     BIGJOB    003000  003000
                                                      .
                                                      .
                                                      .
                                                    CSECT     ILBDSEM0  006268  006268
                                                    *  ENTRY  CURSEGM   00637D<---------(5)
                                                    CSECT     ILBDSRT0  006B38  006B38
                                                      .
                                                      .
                                                      .
      BIGJOB10    0075A8  0075A8  0075E9  64 09 2   CSECT     BIGJOB10  0075A8  0075A8
      BIGJOB47    0075A8  0075A8  007899  65 00 1   CSECT     BIGJOB47  0075A8  0075A8
      BIGJOB48    0075A8  0075A8  0075DB  65 00 2   CSECT     BIGJOB48  0075A8  0075A8
      BIGJOB49    0075A8  0075A8  0075D3  65 01 1   CSECT     BIGJOB49  0075A8  0075A8
      BIGJOB50    0075A8  0075A8  0075F1  65 01 2   CSECT     BIGJOB50  0075A8  0075A8
      BIGJOB60    0075A8  0075A8  0076ED  65 02 1   CSECT     BIGJOB60  0075A8  0075A8
      BIGJOB62    0075A8  0075A8  0075D1  65 02 2   CSECT     BIGJOB62  0075A8  0075A8
      BIGJOB63    0075A8  0075A8  007621  65 03 1   CSECT     BIGJOB63  0075A8  0075A8
      BIGJOB00    0078A0  0078A0  0078A1  65 03 2   CSECT     ILBDDUM0  0078A0<-- 0078A0 (6)
```
Figure 27. Link Editing a Segmented Program

```
              ┌─────────────┐
              │    ROOT     │ Including COBOL subroutines and called programs
              ├─────────────┤
             ┌│  TRANSIENT  │
          L  ┤│    AREA     │ Overlayable and independent segments
             └├─────────────┤
              │SORT PROGRAM │
              └─────────────┘

   L = length of the largest segment in bytes.
```
Figure 28. Location of Sort Program in a Segmentation Structure

PART II

A mass storage device is one on which records can be stored in such a way that the location of any one record can be determined without extensive searching. Records can be accessed directly rather than serially.

The recording surface of a mass storage device is divided into many tracks. A track is defined as a circumference of the recording surface. The number of tracks per recording surface and the capacity of a track for each device are shown in Table 9.

Table 9. Recording Capacities of Mass Storage Devices

Device	Capacity
2311	200 tracks per surface; 3625 bytes per track.
2314, 2319	200 tracks per surface; 7294 bytes per track.
2321	100 tracks per strip; 2000 bytes per track.
3330	404 tracks per surface; 13030 bytes per track.
3330-11*	808 tracks per surface; 13030 bytes per track.
3340 Model 35	348 tracks per surface; 8368 bytes per track.
3340 Model 70	696 tracks per surface; 8368 bytes per track.
3350	555 tracks per surface; 19069 bytes per track.

*In the COBOL ASSIGN statement the 3330-11 is specified as 333B.

Each device has some type of access mechanism through which data is transferred to and from the device. The mechanisms are different for each device, but each mechanism contains a number of read/write heads that transfer data as the recording surfaces rotate past them. Only one head can transfer data (either reading or writing) at a time.

FILE ORGANIZATION

Records in a file must be logically organized so that they can be retrieved efficiently for processing. Four methods of organization for mass storage devices are supported by the DOS/VS COBOL compiler: sequential, direct, indexed, and VSAM. VSAM is discussed in the chapter entitled "Virtual Storage Access Method (VSAM)."

SEQUENTIAL ORGANIZATION

In a sequential file, records are organized solely on the basis of their successive physical location in the file. The records are read or updated in the same order in which they appear.

Individual records cannot be located quickly. Records usually cannot be deleted or added unless the entire file is rewritten. This organization is used when most of the records in the file are processed each time the file is used.

DIRECT ORGANIZATION

A file with direct organization is characterized by some predictable relationship between the key of a record and the address of that record on a mass storage device. This relationship is established by the programmer.

Direct organization is generally used for files where the time required to locate individual records must be kept to an absolute minimum, or for files whose characteristics do not permit the use of sequential or indexed organization.

This organization method has considerable flexibility. The accompanying disadvantage is that although the Disk Operating System/Virtual Storage provides the routines to read or write a file of this type, the programmer is largely responsible for the logic and programming required to locate the key of a record and its address on a mass storage device.

Note: Direct organization is not supported on fixed block devices.

INDEXED ORGANIZATION

An indexed file is similar to a sequential file in that rapid sequential processing is possible. The indexes associated with an indexed file also allow quick retrieval of individual records through random access. Moreover, a separate area of the file is set aside for additions; this eliminates the need to rewrite the entire file when adding records, a process that would usually be necessary with a sequentially organized file. Although the added records are not physically in key sequence, the indexes are constructed in such a way that the added records can be quickly retrieved in key sequence, thus making rapid sequential access possible.

In this method of organization, the system has control over the location of the individual records. Since the characteristics of the file are known, most of the mechanics of locating a particular record are handled by the system.

Note: Indexed organization is not supported on fixed block devices.

DATA MANAGEMENT CONCEPTS

The data management facilities of the Disk Operating System Virtual Storage are provided by a group of routines that are collectively referred to as the Input/Output Control System (IOCS). A distinction is made between two types of routines:

1. Physical IOCS (PIOCS) -- the physical input/output routines included in the Supervisor. PIOCS is used by all programs run within the system. It includes facilities for scheduling input/output operations, checking for and handling error conditions related to input/output devices, and handling input/output interruptions to maintain maximum input/output speeds without burdening the programmer's problem program.

2. Logical IOCS (LIOCS) -- the logical input/output routines linked with the programmer's problem program. These routines provide an interface between the programmer's file processing routines and the PIOCS routines.

   LIOCS performs those functions that a programmer needs to locate and access a logical record for processing. A logical record is one unit of information in a file of similar units, for example, one employee's record in a master payroll file, one part-number record in an inventory file, or one customer account record in an account file. One or more logical records may be included in one physical record. LIOCS refers to the routines that perform the following functions:

   a. Blocking and deblocking records

   b. Switching between input/output areas when two areas are specified for a file

   c. Handling end-of-file and end-of-volume conditions

   d. Checking and writing labels

A brief description of functions performed by LIOCS and their relationship to a COBOL program follows.

Whenever COBOL imperative-statements (READ, WRITE, REWRITE, etc.) are used in a program to control the input/output of records in a file, that file must be defined by a DTF (Define The File) or, for VSAM, an ACB (Access Method Control Block). A DTF or ACB is created for each file opened in a COBOL program from information specified in the Environment Division, FD entry, and input/output statements in the source program. The DTF for each file is part of the object module that is generated by the compiler. The ACB is generated at object time. They describe the characteristics of the logical file, indicate the type of processing to be used for the file, and specify the storage areas and routines used for the file. Further and more detailed onformation in VSAM is to be found in the chapter "VSAM."

One of the constants in the DTF table is the address of a logic module that is to be used at execution time to process that file. A logic module contains the coding necessary to perform data management functions required by the file such as blocking and deblocking, initiating label checking, etc.

Generally, these logic modules are assembled separately and cataloged in the relocatable library under a standard name. At link edit time, the Linkage Editor searches the relocatable library using the virtual reference to locate the logic module. The logic module is then included as part of the program phase. Note that since the Autolink feature of the Linkage Editor is responsible for including the logic modules, the COBOL programmer need not specify any INCLUDE statements.

The type of DTF table prepared by the compiler depends on the organization of the file and the device to which it is assigned. The DTF's used for processing files assigned to mass storage devices are as follows:

DTFSD --	Sequential organization, sequential access
DTFDA --	Direct organization, sequential or random access
DTFIS --	Indexed organization,

For a 3540 diskette unit, the DTF is DTFDU. More detail on this is given in the chapter "Processing 3540 Diskette Unit Files."

The remainder of this chapter provides information about preparing programs which process files assigned to mass storage devices. Included are general descriptions of the organization, the COBOL statements that must be specified in order to build the correct DTF tables, and coding examples.

SEQUENTIAL ORGANIZATION (DTFSD)

In a sequential file on a mass storage device, records are written one after another -- track by track, cylinder by cylinder -- at successively higher addresses.

Records may be fixed-length, spanned, or variable-length, blocked or unblocked, or undefined. Since the file is always accessed sequentially, it is not formatted with keys.

Processing a sequentially organized file for selected records is inefficient. If it is done infrequently, the time spent in locating the records is not significant. The slowest way is to read the records sequentially until the desired one is located. On the average, half of the file must be read to locate one record.

Additions and deletions require a complete rewrite of a sequentially organized file on a mass storage device. Sequential organization is used on mass storage devices primarily for tables and intermediate storage rather than for master files.

Sequentially organized files formatted with keys cannot be created using DTFSD. DTFDA may be used to create and access (sequentially or randomly) such files.

PROCESSING A SEQUENTIALLY ORGANIZED FILE

To create, retrieve, or update a DTFSD file, the following specifications should be made in the source program:

ENVIRONMENT DIVISION

Required clauses:

```
   SELECT [OPTIONAL] file-name

                                  (2311)
                                  (2314)
                                  (2321)
                      (UT)        (2319)
   ASSIGN TO SYSnnn- {    }   -  { 3330 }-S
                      (DA)        (333B)
                                  (3340)
                                  (3350)
                                  (FBA1)
```

Optional clauses:

```
   RESERVE Clause
   FILE-LIMIT Clause
   ACCESS MODE IS SEQUENTIAL
   PROCESSING MODE IS SEQUENTIAL
   RERUN Clause
   SAME Clause
   APPLY WRITE-ONLY Clause (create only)
   APPLY WRITE-VERIFY Clause (create or
     update only)
```

Invalid clauses:

```
   ACCESS MODE IS RANDOM
   ACTUAL KEY Clause
   NOMINAL KEY Clause
   RECORD KEY Clause
   TRACK-AREA Clause
   MULTIPLE FILE TAPE Clause
   APPLY EXTENDED-SEARCH Clause
   APPLY CYL-OVERFLOW Clause

         (MASTER-INDEX)
   APPLY {            } Clause
         (CYL-INDEX   )

   APPLY CORE-INDEX Clause
```

DTFSD files may be opened as INPUT, OUTPUT, or I-O. When creating such a file, an INVALID KEY condition occurs when the file limit has been reached and an attempt is made to place another record on the mass storage device. The file limit is determined from the EXTENT control statements.

When a DTFSD file is opened as OUTPUT, each WRITE statement signifies the creation of a new record. When opened as I-O, each WRITE statement signifies that the record just read is to be rewritten.

DIRECT ORGANIZATION (DTFDA)

With direct organization, there is a definite relationship between the key of a record and its address. This relationship permits rapid access to any record if the file is carefully organized. The programmer develops a record address that ranges from zero to some maximum by converting a particular field in each record to a track address. Each byte in the address is a binary number. To reference a particular record, the programmer must supply both the track address and the identifier that makes each record unique on its track. Both the track address and the identifier are supplied by the programmer in the ACTUAL KEY clause. This will be discussed in detail later in this chapter.

With direct organization, records may be fixed length, spanned or undefined. The records must be unblocked. R0 (record zero) of each track is used as a capacity record. It contains the address of the last record written on the track, and is used by the system to determine whether a new record will fit on the track. The capacity records are updated by the system as records are added to the file. The capacity records do not account for deletions: as far as the system is concerned, once a track is full it remains full (even if the programmer deletes records) until the file is reorganized.

Often, more records are converted to a given track address than will actually fit on the track. These surplus records are known as overflow records and are usually written into a separate area known as an overflow area.

As already noted, the programmer has an unlimited choice in deciding where records are to be located in a directly organized file. The logic and programming are his responsibility.

When creating or making additions to the file, the programmer must specify the location for a record (track address) and the identifier that makes each record on the track unique. If there is space on the track, the system writes the record and updates the capacity record. If the specified track is full, a standard error condition occurs, and the programmer may specify another track address in his USE AFTER STANDARD ERROR declarative routine.

In the case of one maximum size record per track (when spanned records are not specified), the data length plus the length of the symbolic key cannot exceed the following values:

2311 -- 3605 bytes
2314, 2319 -- 7249 bytes
2321 -- 1984 bytes
3330 -- 12974 bytes
3340 -- 8293 bytes
3350 -- 18987 bytes

When reading or updating the file, the programmer must supply the track address and the unique identifier on the track for the specific record being sought. The system locates the track and searches that track for the record with the specified identifier. If the record is not found, COBOL indicates this to the programmer by raising an INVALID KEY condition. Only the track specified by the programmer is searched. If EXTENDED-SEARCH is applied, the search for a specified record key begins on the track specified and continues until one of two conditions occurs:

1. The record is found.

2. The end of the specified cylinder is reached.

In the second case, the INVALID-KEY option of the READ or REWRITE is executed. To ensure file integrity, the upper limit of each extent of a file using EXTENDED-SEARCH must be the last track of a cylinder.

Error recovery from a DTFDA file is described in detail in the chapter "Advanced Processing Capabilities."

ACCESSING A DIRECTLY ORGANIZED FILE

A directly organized file (DTFDA) may be accessed either sequentially or randomly.

<u>ACCESSING A DIRECTLY ORGANIZED FILE SEQUENTIALLY</u>: When reading a direct file sequentially, records are retrieved in logical sequence; this logical sequence corresponds exactly to the physical sequence of the records. To retrieve a DTFDA file sequentially, the following specifications are made in the source program:

<u>ENVIRONMENT DIVISION</u>

<u>Required clauses</u>:

SELECT [OPTIONAL] file-name

$$\text{ASSIGN TO SYSnnn-DA-}\left\{\begin{matrix}2311\\2321\\2314\\2319\\3330\\333B\\3340\\3350\end{matrix}\right\}\text{-}\left\{\begin{matrix}A\\D\end{matrix}\right\}$$

<u>Optional clauses</u>:

FILE-LIMIT Clause
ACCESS MODE IS SEQUENTIAL
PROCESSING MODE IS SEQUENTIAL
ACTUAL KEY Clause
RERUN Clause
SAME Clause

Invalid clauses:

```
RESERVE Clause
ACCESS MODE IS RANDOM
NOMINAL KEY Clause
RECORD KEY Clause
TRACK-AREA Clause
MULTIPLE FILE TAPE Clause
APPLY WRITE-ONLY Clause
```

APPLY CYL-OVERFLOW Clause
APPLY EXTENDED- SEARCH Clause
APPLY WRITE-VERIFY Clause

$$\text{APPLY}\left\{\begin{array}{l}\text{MASTER-INDEX}\\ \text{CYL-INDEX}\end{array}\right\}\text{Clause}$$

APPLY CORE-INDEX Clause

When DTFDA records are retrieved sequentially, the file may be opened only as INPUT. The AT END condition occurs when the last record has been read and execution of another READ is attempted.

Note that in the ASSIGN clause, an $\underline{A}$ must be specified for files with actual track addressing, and a $\underline{D}$ must be specified for files with relative track addressing.

<u>ACCESSING A DIRECTLY ORGANIZED FILE RANDOMLY</u>: To create a directly organized file randomly, the following specifications are made in the source program:

<u>ENVIRONMENT DIVISION</u>

<u>Required clauses</u>:

SELECT file-name

$$\text{ASSIGN TO SYSnnn-DA-}\left\{\begin{array}{l}2311\\2321\\2314\\2319\\3330\\333B\\3340\\3350\end{array}\right\}\text{-}\left\{\begin{array}{l}\text{A}\\ \text{D}\end{array}\right\}$$

ACCESS MODE IS RANDOM
ACTUAL KEY Clause

<u>Optional clauses</u>:

FILE-LIMIT Clause
PROCESSING MODE IS SEQUENTIAL
RERUN Clause
SAME Clause
APPLY WRITE-VERIFY Clause

<u>Invalid clauses</u>:

RESERVE Clause
ACCESS MODE IS SEQUENTIAL
NOMINAL KEY Clause
RECORD KEY Clause
TRACK-AREA Clause
MULTIPLE FILE TAPE Clause
APPLY WRITE-ONLY Clause
APPLY EXTENDED-SEARCH Clause
APPLY WRITE-VERIFY Clause
APPLY CYL-OVERFLOW Clause

$$\text{APPLY}\left\{\begin{array}{l}\text{MASTER-INDEX}\\ \text{CYL-INDEX}\end{array}\right\}\text{Clause}$$

APPLY CORE-INDEX Clause

Note that in the ASSIGN clause, an $\underline{A}$ must be specified for files with actual track addressing, and a $\underline{D}$ must be specified for files with relative track addressing.

To retrieve or update a directly organized file randomly, the following specifications must be made in the source program.

<u>ENVIRONMENT DIVISION</u>

<u>Required clauses</u>:

SELECT file-name

$$\text{ASSIGN TO SYSnnn-DA-}\left\{\begin{array}{l}2311\\2314\\2321\\2319\\3330\\333B\\3340\\3350\end{array}\right\}\text{-}\left\{\begin{array}{l}\text{A}\\ \text{D}\\ \text{U}\\ \text{W}\end{array}\right\}$$

ACCESS MODE IS RANDOM
ACTUAL KEY Clause

Note that in the ASSIGN clause an $\underline{A}$ must be specified for files with actual track addressing, a $\underline{D}$ must be specified for files with relative track addressing, a $\underline{U}$ must be specified for files with actual track addressing when the REWRITE statement is used, and $\underline{W}$ must be specified for files with relative track addressing when the REWRITE statement is used.

The optional and invalid clauses are the same as those specified previously for creating a directly organized file.

<u>Exception</u>: APPLY EXTENDED-SEARCH is optional when retrieving or updating a directly organized file randomly.

ACTUAL KEY CLAUSE

Note that the ACTUAL KEY clause is required for DTFDA files when ACCESS IS RANDOM, is optional for DTFDA files when ACCESS IS SEQUENTIAL, and is not used for DTFSD files.

The actual key consists of two components. One component expresses the track address at which the record is to be placed for an output operation, or at which the search is to begin for an input operation. The track address can be expressed either as an actual address or as a relative address, depending upon the addressing scheme chosen when the file was created. The other component is associated with the record itself and serves as its unique identifier. The structures of both actual keys are shown in Figure 29.

```
          Actual Key
     Actual Track | Record Identifier
     Address      |
Byte 1          8 9               263

          Actual Key
     Relative     | Record Identifier
     Track Address|
Byte 1          4 5               258
```

Figure 29. Structures of the Actual Key

The format of the ACTUAL KEY clause is:

<u>ACTUAL</u> KEY <u>IS</u> data-name

When actual track addressing is used, <u>data-name</u> may be any fixed item from 9 through 263 bytes in length. It must be defined in the Working-Storage, File, or Linkage Section. The first eight bytes are used to specify the actual track address. The structure of these bytes and permissible specifications for the mass storage devices are shown in Figure 30. The programmer may select from 1 to 255 bytes for the record identifier portion of the actual key field.

<u>Note</u>: If a SEEK statement is used when retrieving a direct file randomly, actual track addressing is required.

When relative track addressing is used, data-name may be any fixed item from 5 through 258 bytes in length. It must be defined in the File Section, the Working-Storage Section, or the Linkage Section. The first four bytes of data-name are the track identifier. The identifier is used to specify the relative track address for the record and must be defined as an 8-integer binary data item whose maximum value does not exceed 16,777,215. The remainder of data-name, which is 1 through 254 bytes in length, is the record identifier. It represents the symbolic portion of the key field used to identify a particular record on a track.

For a complete discussion of the ACTUAL KEY clause, see the publication <u>IBM DOS Full American National Standard COBOL</u>.

<u>Randomizing Techniques</u>

One method of determining the value of the track address portion of the field defined in the ACTUAL KEY clause is referred to as <u>indirect addressing</u>. Indirect addressing generally is used when the range of keys for a file includes a high percentage of unused values. For example, employee numbers may range from 000001 to 009999, but only 3000 of the possible 9999 numbers are currently assigned. Indirect addressing is also used for nonnumeric keys. Key, in this discussion, refers to that field of the record being written that will be converted to the track address portion.

Indirect addressing signifies that the key is converted to a value for the actual track address by using some algorithm intended to limit the range of addresses.

	Pack	Cell		Cylinder		Head		Record
	M	B	B	C	C	H	H	R
Byte / Device	0	1	2	3	4	5	6	7
2311	0-221	0	0	0	0-199	0	0-9	0-255
2314	0-221	0	0	0	0-199	0	0-19	0-255
2321	0-221	0	0-9	0-19	0-9	0-4	0-19	0-255
3330	0-221	0	0	0-403		0	0-18	0-255
3330-11	0-221	0	0	0-807		0	0-18	0-255
3340 Model 35	0-221	0	0	0-347		0	0-11	0-255
3340 Model 70	0-221	0	0	0-695		0	0-11	0-255
3350	0-221	0	0	0-554		0	0-29	0-255

Figure 30. Permissible Specifications for the First Eight Bytes of the Actual Key

Such an algorithm is called a randomizing technique. Randomizing techniques need not produce a unique address for every record and, in fact, such techniques usually produce synonyms. Synonyms are records whose keys randomize to the same address.

Two objectives must be considered in selecting a randomizing technique:

1. Every possible key in the file must randomize to an address within the designated range.

2. The addresses should be distributed evenly across the range so that there are as few synonyms as possible.

Note that one way to minimize synonyms is to allocate more space for the file than is actually required to contain all the records. For example, the percentage of locations that are actually used might be 80% to 85% of the allocated space.

When actual track addressing is used, the first eight bytes of the ACTUAL KEY field can be thought of as a "discontinuous binary address." This is significant to the programmer because he must keep two considerations in mind. First, the cylinder and head number must be in binary notation, so the results of the randomizing formula must be in binary format. Second, the address is "discontinuous" since a mathematical overflow from one element (e.g., head number) does not increment the adjacent element (e.g., cylinder number).

DIVISION/REMAINDER METHOD: One of the simplest ways to indirectly address a directly organized file is by using the division/remainder method. (For a discussion of other randomizing techniques, see the publication Introduction to IBM Direct Access Storage Devices and Organization Methods, Order No. GC20-1649.)

1. Determine the amount of locations required to contain the data file. Include a packing factor for additional space to eliminate synonyms. The packing factor should be approximately 20% of the total space allocated to contain the data file.

2. Select, from the prime number table, the nearest prime number that is less than the total of step 1. A prime number is a number divisible only by itself and the integer 1. Table 10 is a partial list of prime numbers.

3. Clear any zones from the first eight bytes of the actual key field. This can be accomplished by moving the key to a field described as COMPUTATIONAL.

4. Divide the key by the prime number selected.

5. Ignore the quotient; utilize the remainder as the relative location within the data file.

6. (For actual track addressing only) Locate the beginning of the space available and manipulate the relative address, to the actual device address if necessary.

For example, assume that a company is planning to create an inventory file on a 2311 disk storage device. There are 8000 different inventory parts, each identified by an 8-character part number. Using a 20% packing factor, 10,000 record positions are allocated to store the data file.

Method A: The closest prime number to 10,000, but under 10,000, is 9973. Using one inventory part number as an example, in this case #25DF3514, and clearing the zones we have 25463514. Dividing by 9973 we get a quotient of 2553 and a remainder of 2445. 2445 is the relative location of the record within the data file corresponding to part number 25DF3514. The record address can be determined from the relative location as follows:

1. (For actual track addressing only) Determine the beginning point for the data file (e.g., cylinder 100, track 0).

2. Determine the number of records that can be stored on a track (e.g., twelve per track on a 2314 disk pack, assuming each inventory record is 200 bytes long).

   Because each data record contains non-data components, such as a count area and interrecord gaps, track capacity for data storage will vary with record length. As the number of separate records on a track increases, interrecord gaps occupy additional byte positions so that data capacity is reduced. Track capacity formulas provide the means to determine total byte requirements for records of various sizes on a track. These formulas can be found in the publications IBM Component Descriptions, Order Nos. GA26-5988 and GA26-3599.

3. Divide the relative number (2445) by the number of records to be stored on each track.

4. (For actual track addressing only) The result, quotient = 203, is now divided into cylinder and head designation. Since the 2311 disk pack has ten heads, the quotient of 203 is divided by 10 to show:

   Cylinder or CC = 20
   Head or HH = 03 (high-order zero added)

4B. (For relative track addressing only) The result, quotient = 203, now becomes the track identifier of the actual key.

Method B: Utilizing the same example, another approach will also provide the relative track address:

1. The number of records that may be contained on one track is twelve. Therefore, if 10,000 record locations are to be provided, 834 tracks must be reserved.

2. The prime number nearest, but less than 834, is 829.

3. Divide the zone-stripped key by the prime value. (In the example, 25463514 divided by 829 provides a quotient of 30715 and a remainder of 779. The remainder is the relative address.)

Table 10. Partial List of Prime Numbers (Part 1 of 2)

Table 10. Partial List of Prime Numbers (Part 2 of 2)

A (Number)	B (Nearest Prime Number Less Than A)
500	499
600	599
700	691
800	797
900	887
1000	997
1100	1097
1200	1193
1300	1297
1400	1399
1500	1499
1600	1597
1700	1699
1800	1789
1900	1889
2000	1999
2100	2099
2200	2179
2300	2297
2400	2399
2500	2477
2600	2593
2700	2699
2800	2797
2900	2897
3000	2999
3100	3089
3200	3191
3300	3299
3400	3391
3500	3499
3600	3593
3700	3697
3800	3797
3900	3889
4000	3989
4100	4099
4200	4177
4300	4297
4400	4397
4500	4493
4600	4597
4700	4691
4800	4799
4900	4889
5000	4999
5100	5099
5200	4197
5300	5297
5400	4399
5500	5483
5600	5591
5700	5693
5800	5791
5900	5897
6000	5987
6100	6091
6200	6199
6300	6299
6400	6397
6500	6491
6600	6599
6700	6691
6800	6793
6900	6899
7000	6997
7100	7079
7200	7193
7300	7297
7400	7393
7500	7499
7600	7591
7700	7699
7800	7793
7900	7883
8000	7993
8100	8093
8200	8191
8300	8297
8400	8389
8500	8467
8600	8599
8700	8699
8800	8793
8900	8899
9000	8899
9100	9091
9200	9199
9300	9293
9400	9397
9500	9497
9600	9587
9700	9697
9800	9791
9900	9887
10,000	9973
10,100	10,099
10,200	10,193
10,300	10,289
10,400	10,399
10,500	10,499
10,600	10,597

4. (For actual track addressing only) To convert the relative address to an actual device address, divide the relative address by the number of tracks in a cylinder. The quotient will provide the cylinder number and the remainder will be the track number. For example, the 2311 disk pack would utilize 779 as:

   Cylinder or CC = 77
   Track or HH = 9

Figure 31 is a sample COBOL program which creates a direct file with actual track addressing using Method B and provides for the possibility of synonym overflow. Synonym overflow will occur if a record randomizes to a track that is already full. The following description highlights the features of the example. Circled numbers on the program listing correspond to the numbers in the text.

(1) The value 10 is added to TRACK-1 to ensure that the problem program does not write on cylinder 0. Cylinder 0 must be reserved for the Volume Table of Contents.

- Since the prime number used as a divisor is 829, the largest possible remainder will be 828. Adding 10 to TRACK-1 adjusts the largest possible remainder to 838.

(2) If synonym overflow occurs, control is given to the error procedure declarative specified in the first section of the Procedure Division. The declarative provides that:

- Any record which cannot fit on a track (i.e., tracks 0 through 8 of any cylinder) will be written in the first available position on the following track(s).

- Any record which cannot fit within a single cylinder will be written on cylinder 84 (i.e., the cylinder overflow area).

- If a record cannot fit on either cylinders 1 through 83, or on cylinder 84, the job is terminated.

(3) The standard error condition "no room found" is tested before control is given to the synonym routine. Other standard error conditions as well as invalid key conditions result in job termination.

ERROR-COND is the identifier which specifies the error condition that caused control to be given to the error declarative. ERROR-COND is printed on SYSLST whenever the error declarative section is entered. TRACK-ID and C-REC are also printed on SYSLST. They are printed before the execution of each WRITE statement. This output has been provided in order to facilitate an understanding of the logic involved in the creation of D-FILE.

(4) The first twelve records which randomize to cylinder 002 track 8 are actually written on track 8.

(5) The next twelve records which randomize to cylinder 002 track 8 are adjusted by the SYNONYM-ROUTINE and written on cylinder 002 track 9.

(6) The next twelve records which randomize to cylinder 002 track 8 are adjusted by the SYNONYM-ROUTINE and written on cylinder 84 track 0 (i.e., the overflow cylinder).

(7) The last two records which randomize to cylinder 002 track 8 are adjusted by the SYNONYM-ROUTINE and written on cylinder 84 track 1 (i.e., the overflow cylinder).

```
// JOB METHODBA
// OPTION NODECK,LINK,LIST,LISTX,SYM,ERRS
// EXEC FCOBOL
```

```
  1  IBM DOS VS COBOL                          REL 1.0          PP NO. 5746-CB1                 08.47.44  10/04/73

          IDENTIFICATION DIVISION.
          PROGRAM-ID. METHOD-B.
          ENVIRONMENT DIVISION.
          CONFIGURATION SECTION.
          SOURCE-COMPUTER. IBM-370.
          OBJECT-COMPUTER. IBM-370.
          INPUT-OUTPUT SECTION.
          FILE-CONTROL.
              SELECT D-FILE ASSIGN SYS015-DA-2314-A-MASTER
              ACCESS IS RANDOM
              ACTUAL KEY IS ACT-KEY.
              SELECT C-FILE ASSIGN TO SYS007-UR-2540R-S.
          DATA DIVISION.
          FILE SECTION.
          FD  D-FILE
              LABEL RECORDS ARE STANDARD.
          01  D-REC.
                  02  PART-NUM PIC X(8).
                  02  NUM-ON-HAND PIC 9(4).
                  02  PRICE PIC 9(5)V99.
                  02  FILLER PIC X(181).
          FD  C-FILE
              LABEL RECORDS ARE OMITTED.
          01  C-REC.
                  02  PART-NUM PIC X(8).
                  02  NUM-ON-HAND PIC 9(4)9.
                  02  PRICE PIC 9(5)V99.
          WORKING-STORAGE SECTION.
          77  HD     PIC 9 VALUE ZERO.
          77  SAVE PIC S9(8) COMP SYNC.
          77  QUOTIENT PIC S9(5) COMP SYNC.
          01  ERROR-COND.
              02  FILLER PIC 99  VALUE ZERO.
              02  ERR     PIC 9  VALUE ZERO.
              02  FILLER PIC 9(5)  VALUE ZERO.
          01  TRACK-1     PIC 9999.
          01  TRACK-ID REDEFINES TRACK-1.
              02  CYL PIC 999.
              02  HEAD PIC 9.
          01  KEY-1.
              02  M    PIC S999 COMP SYNC VALUE ZEROES.
              02  BB   PIC S9  COMP SYNC VALUE ZERO.
              02  CC   PIC S999 COMP SYNC.
              02  HH   PIC S999   COMP SYNC.
              02  R    PIC X   VALUE LOW-VALUE.
              02  REC-ID   PIC X(8).
          01  KEY-2 REDEFINES KEY-1.
              02  FILLER PIC X.
              02  ACT-KEY  PIC X(16).
```

Figure 31. Creating a Direct File Using Method B (Part 1 of 4)

```
IBM DOS VS COBOL                 REL 1.0              PP NO. 5746-CB1          08.47.44     10/04/73

       PROCEDURE DIVISION.
       DECLARATIVES.
       ERROR-PROCEDURE SECTION. USE AFTER STANDARD ERROR PROCEDURE
               ON D-FILE GIVING ERROR-COND.




       ERROR-ROUTINE.
           EXHIBIT NAMED ERROR-COND.
               IF ERR = 1 GO TO SYNONYM-ROUTINE   ELSE                3
                   DISPLAY 'OTHER STANDARD ERROR'   REC-ID
               GO TO EOJ.                                                  2
       SYNONYM-ROUTINE.
           IF CC = 84 AND HD = 9 DISPLAY 'OVERFLOW AREA FULL'
               GO TO EOJ.
           IF CC = 84 ADD 1 TO HD  GO TO ADJUST-HD.
           IF HH = 9 GO TO END-CYLINDER.
           ADD 1 TO HH.
           GO TO WRITES.
       END-CYLINDER.
           MOVE 84 TO CC.
       ADJUST-HD.
           MOVE HD TO HH.
           GO TO WRITES.
       END DECLARATIVES.
       FILE-CREATION SECTION.
           OPEN INPUT C-FILE
               OUTPUT D-FILE.
       READS.
           READ C-FILE AT END GO TO EOJ.
           MOVE CORRESPONDING C-REC TO D-REC.
           MOVE PART-NUM OF C-REC TO REC-ID SAVE.
           DIVIDE SAVE BY 829 GIVING QUOTIENT REMAINDER TRACK-1.   1
           ADD 10 TO TRACK-1.
           MOVE CYL TO CC.
           MOVE HEAD TO HH.
       WRITES.
           EXHIBIT NAMED TRACK-ID  C-REC  CC  HH.
           WRITE D-REC INVALID KEY GO TO INVALID-KEY.
           GO TO READS.
       INVALID-KEY.
           DISPLAY 'INVALID KEY'   REC-ID.
       EOJ.
           CLOSE C-FILE D-FILE.
           STOP RUN.


// LBLTYP NSD(01)
// EXEC LNKEDT
```

Figure 31. Creating a Direct File Using Method B (Part 2 of 4)

```
        IBM DOS VS COBOL                           REL 1.0              PP NO.5746-CB1              08.47.44     10/04/73

// ASSGN SYS007,X'00C'
// ASSGN SYS015,X'231'
// DLBL MASTER,,99/365,DA
// EXTENT SYS015,111111,1,0,20,840
// EXEC

TRACK-ID = 0010 C-REC = 82900000           CC = 001 HH = 000
TRACK-ID = 0011 C-REC = 82900001           CC = 001 HH = 001
TRACK-ID = 0028 C-REC = 8290001801         CC = 002 HH = 008
TRACK-ID = 0028 C-REC = 8290001802         CC = 002 HH = 008
TRACK-ID = 0028 C-REC = 8290001803         CC = 002 HH = 008
TRACK-ID = 0028 C-REC = 8290001804         CC = 002 HH = 008
TRACK-ID = 0028 C-REC = 8290001805         CC = 002 HH = 008     4
TRACK-ID = 0028 C-REC = 8290001806         CC = 002 HH = 008
TRACK-ID = 0028 C-REC = 8290001807         CC = 002 HH = 008
TRACK-ID = 0028 C-REC = 8290001808         CC = 002 HH = 008
TRACK-ID = 0028 C-REC = 8290001809         CC = 002 HH = 008
TRACK-ID = 0028 C-REC = 8290001810         CC = 002 HH = 008
TRACK-ID = 0028 C-REC = 8290001811         CC = 002 HH = 008
TRACK-ID = 0028 C-REC = 8290001812         CC = 002 HH = 008
TRACK-ID = 0028 C-REC = 8290001813         CC = 002 HH = 008
TRACK-ID = 0028 C-REC = 8290001814         CC = 002 HH = 008
TRACK-ID = 0186 C-REC =  290001815         CC = 018 HH = 006
TRACK-ID = 0186 C-REC =  290001816         CC = 018 HH = 006
TRACK-ID = 0028 C-REC = 8290001817         CC = 002 HH = 008
TRACK-ID = 0028 C-REC = 8290001818         CC = 002 HH = 008
TRACK-ID = 0028 C-REC = 8290001819         CC = 002 HH = 008
TRACK-ID = 0028 C-REC = 8290001820         CC = 002 HH = 008     5
TRACK-ID = 0028 C-REC = 8290001821         CC = 002 HH = 008
TRACK-ID = 0028 C-REC = 8290001822         CC = 002 HH = 008
TRACK-ID = 0028 C-REC = 8290001823         CC = 002 HH = 008
ERROR-COND = 00100000
TRACK-ID = 0028 C-REC = 8290001823         CC = 002 HH = 009
TRACK-ID = 0028 C-REC = 8290001824         CC = 002 HH = 008
ERROR-COND = 00100000
TRACK-ID = 0028 C-REC = 8290001824         CC = 002 HH = 009
```

Figure 31. Creating a Direct File Using Method B (Part 3 of 4)

```
 IBM DOS VS COBOL                          REL 1.0        PP NO. 5746-CB1                08.47.44     10/04/73

TRACK-ID = 0028 C-REC = 8290001825          CC = 002 HH = 008
ERROR-COND = 00100000
TRACK-ID = 0028 C-REC = 8290001825          CC = 002 HH = 009
TRACK-ID = 0028 C-REC = 8290001826          CC = 002 HH = 008
ERROR-COND = 00100000
TRACK-ID = 0028 C-REC = 8290001826          CC = 002 HH = 009
TRACK-ID = 0011 C-REC =  8290001827         CC = 001 HH = 001
TRACK-ID = 0011 C-REC =  8290001828         CC = 001 HH = 001
TRACK-ID = 0011 C-REC =  8290001829         CC = 001 HH = 001
TRACK-ID = 0028 C-REC = 8290001830          CC = 002 HH = 008
ERROR-COND = 00100000
TRACK-ID = 0028 C-REC = 8290001830          CC = 002 HH = 009
TRACK-ID = 0028 C-REC = 8290001831          CC = 002 HH = 008
ERROR-COND = 00100000
TRACK-ID = 0028 C-REC = 8290001831          CC = 002 HH = 009        6
TRACK-ID = 0028 C-REC = 8290001832          CC = 002 HH = 008
ERROR-COND = 00100000
TRACK-ID = 0028 C-REC = 8290001832          CC = 002 HH = 009
TRACK-ID = 0028 C-REC = 8290001833          CC = 002 HH = 008
ERROR-COND = 00100000
TRACK-ID = 0028 C-REC = 8290001833          CC = 002 HH = 009
TRACK-ID = 0028 C-REC = 8290001834          CC = 002 HH = 008
ERROR-COND = 00100000
TRACK-ID = 0028 C-REC = 8290001834          CC = 002 HH = 009
TRACK-ID = 0028 C-REC = 8290001835          CC = 002 HH = 008
ERROR-COND = 00100000
TRACK-ID = 0028 C-REC = 8290001835          CC = 002 HH = 009
TRACK-ID = 0028 C-REC = 8290001836          CC = 002 HH = 008
ERROR-COND = 00100000
TRACK-ID = 0028 C-REC = 8290001836          CC = 002 HH = 009
TRACK-ID = 0028 C-REC = 8290001837          CC = 002 HH = 008
ERROR-COND = 00100000
TRACK-ID = 0028 C-REC = 8290001837          CC = 002 HH = 009        7
TRACK-ID = 0028 C-REC = 8290001838          CC = 002 HH = 008
ERROR-COND = 00100000
TRACK-ID = 0028 C-REC = 8290001838          CC = 002 HH = 009
```

Figure 31. Creating a Direct File Using Method B (Part 4 of 4)

Figure 32 is a sample COBOL program which creates a direct file with relative track addressing using Method B. The sample program provides for the possibility of synonym overflow. Synonym overflow will occur if a record randomizes to a track which is already full. The following discussion highlights some basic features. Circled numbers on the program listing correspond to numbers in the text.

① Since the prime number used as a divisor is 829, the largest possible remainder will be 828.

② If synonym overflow occurs, control is given to the USE AFTER STANDARD ERROR declarative specified in the first section of the Procedure Division. The declarative provides that any record that cannot fit on the track to which it randomizes will be written on the first subsequent track available.

③ The standard error condition "no room found" is tested before control is given to the SYNONYM-ROUTINE. Other standard error conditions as well as invalid key conditions result in job termination (EOJ).

ERROR-COND is the identifier which specifies the error condition that caused control to be given to the error declarative. ERROR-COND is printed on SYSLST whenever the error declarative section is entered. TRACK-ID and C-REC are also printed on SYSLST before execution of each WRITE statement. This output has been provided in order to facilitate an understanding of the logic involved in the creation of D-FILE.

④ The first twelve records which randomize to relative track 18 are actually written on relative track 18.

⑤ The next twelve records which randomize to relative track 18 are adjusted by the SYNONYM-ROUTINE and are actually written on relative track 19.

⑥ The next twelve records which randomize to relative track 18 are adjusted by the SYNONYM-ROUTINE and are actually written on relative track 20.

⑦ The last two records which randomize to relative track 18 are adjusted by the SYNONYM-ROUTINE and are actually written on relative track 21.

```
// JOB METHODBR
// OPTION NODECK,LINK,LIST,LISTX,SYM,ERRS
// EXEC FCOBOL




    1  IBM DOS VS COBOL                             REL 1.0           PP NO. 5746-CB1                 08.40.53  10/04/73

 CBL QUOTE
              IDENTIFICATION DIVISION.
              PROGRAM-ID. METHODB.
              ENVIRONMENT DIVISION.
              CONFIGURATION SECTION.
              SOURCE-COMPUTER. IBM-370.
              OBJECT-COMPUTER. IBM-370.
              INPUT-OUTPUT SECTION.
              FILE-CONTROL.
                  SELECT D-FILE ASSIGN TO SYS015-DA-2314-D-MASTER
                  ACCESS IS RANDOM
                  ACTUAL KEY IS ACT-KEY.
                  SELECT C-FILE ASSIGN TO SYS007-UR-2540R-S.
              DATA DIVISION.
              FILE SECTION.
              FD  D-FILE
                  LABEL RECORDS ARE STANDARD.
              01  D-REC.
                  05 PART-NUM PIC X(8).
                  05 NUM-ON-HAND PIC 9(4).
                  05 PRICE PIC 9(5)V99.
                  05 FILLER PIC X(181).
              FD  C-FILE
                  LABEL RECORDS ARE OMITTED.
              01  C-REC.
                  05 PART-NUM PIC X(8).
                  05 NUM-ON-HAND PIC 9(4).
                  05 PRICE PIC 9(5)V99.
                  05 FILLER PIC X(61).
              WORKING-STORAGE SECTION.
              77  SAVE PIC S9(8) COMP SYNC.
              77  QUOTIENT PIC S9(8) COMP SYNC.
              01  ACT-KEY.
                  02 TRACK-ID PIC S9(8) COMP SYNC.
                  02 REC-ID PIC X(8).
              01  ERROR-COND.
                  02 FILLER PIC 99 VALUE ZERO.
                  02 ERR PIC 9 VALUE ZERO.
                  02 FILLER PIC 9(5) VALUE ZERO.
```

Figure 32. Creating a Direct File with Relative Track Addressing Using Method B (Part 1 of 4)

```
IBM DOS VS COBOL                    REL 1.0          PP NO. 5746-CB1           08.40.53    10/04/73

          PROCEDURE DIVISION.
          DECLARATIVES.
          ERROR-PROCEDURE SECTION. USE AFTER STANDARD ERROR PROCEDURE
              ON D-FILE GIVING ERROR-COND.
          ERROR-ROUTINE.
              EXHIBIT NAMED ERROR-COND.
              IF ERR = 1 GO TO SYNONYM-ROUTINE ELSE             ③        ②
                  DISPLAY "OTHER STANDARD ERROR  " REC-ID
                  GO TO EOJ.
          SYNONYM-ROUTINE.
              IF TRACK-ID IS LESS THAN 834, ADD 1 TO TRACK-ID. GO TO
                  WRITES.
          END DECLARATIVES.
              OPEN INPUT C-FILE

                   OUTPUT D-FILE.
          READS.
              READ C-FILE AT END GO TO EOJ.
              MOVE CORRESPONDING C-REC TO D-REC.
              MOVE PART-NUM OF C-REC TO REC-ID, SAVE.                 ①
              DIVIDE SAVE BY 829 GIVING QUOTIENT REMAINDER TRACK-ID.
          WRITES.
              EXHIBIT NAMED TRACK-ID C-REC.
              WRITE D-REC INVALID KEY GO TO INVALID-KEY.
              GO TO READS.
          INVALID-KEY.
              DISPLAY "INVALID KEY  " REC-ID.
          EOJ.
              CLOSE C-FILE D-FILE.
              STOP RUN.


// LBLTYP NSD(01)
// EXEC LNKEDT
```

Figure 32. Creating a Direct File with Relative Track Addressing Using Method B (Part 2 of 4)

```
// ASSGN SYS007,X'00C'
// ASSGN SYS015,X'231'
// DLBL MASTER,,99/365,DA
// EXTENT SYS015,111111,1,0,20,840
// EXEC

TRACK-ID = 00000000 C-REC = 82900000
TRACK-ID = 00000001 C-REC = 82900001
TRACK-ID = 00000018 C-REC = 8290001801
TRACK-ID = 00000018 C-REC = 8290001802
TRACK-ID = 00000018 C-REC = 8290001803
TRACK-ID = 00000018 C-REC = 8290001804
TRACK-ID = 00000018 C-REC = 8290001805
TRACK-ID = 00000018 C-REC = 8290001806      ④
TRACK-ID = 00000018 C-REC = 8290001807
TRACK-ID = 00000018 C-REC = 8290001808
TRACK-ID = 00000018 C-REC = 8290001809
TRACK-ID = 00000018 C-REC = 8290001810
TRACK-ID = 00000018 C-REC = 8290001811
TRACK-ID = 00000018 C-REC = 8290001812
TRACK-ID = 00000018 C-REC = 8290001813
TRACK-ID = 00000018 C-REC = 8290001814
TRACK-ID = 00000018 C-REC = 8290001815
TRACK-ID = 00000018 C-REC = 8290001816
TRACK-ID = 00000018 C-REC = 8290001817
TRACK-ID = 00000018 C-REC = 8290001818
TRACK-ID = 00000018 C-REC = 8290001819
TRACK-ID = 00000018 C-REC = 8290001820
TRACK-ID = 00000018 C-REC = 8290001821
ERROR-COND = 00100000                       ⑤
TRACK-ID = 00000019 C-REC = 8290001821
TRACK-ID = 00000018 C-REC = 8290001822
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001822
TRACK-ID = 00000018 C-REC = 8290001823
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001823
TRACK-ID = 00000018 C-REC = 8290001824
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001824
```

Figure 32. Creating a Direct File with Relative Track Addressing Using Method B (Part 3 of 4)

```
IBM DOS VS COBOL                    REL 1.0               PP NO. 5746-CB1          08.40.53   10/04/73

TRACK-ID = 00000018 C-REC = 8290001825
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001825
TRACK-ID = 00000018 C-REC = 8290001826
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001826
TRACK-ID = 00000018 C-REC = 8290001827
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001827
TRACK-ID = 00000018 C-REC = 8290001828
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001828
TRACK-ID = 00000018 C-REC = 8290001829
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001829
TRACK-ID = 00000018 C-REC = 8290001830
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001830      6
TRACK-ID = 00000018 C-REC = 8290001831
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001831
TRACK-ID = 00000018 C-REC = 8290001832
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001832
TRACK-ID = 00000018 C-REC = 8290001833
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001833
TRACK-ID = 00000018 C-REC = 8290001834
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001834
TRACK-ID = 00000018 C-REC = 8290001835
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001835
TRACK-ID = 00000018 C-REC = 8290001836
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001836
TRACK-ID = 00000018 C-REC = 8290001837
ERROR-COND = 00100000                    7
TRACK-ID = 00000019 C-REC = 8290001837
TRACK-ID = 00000018 C-REC = 8290001838
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001838
```

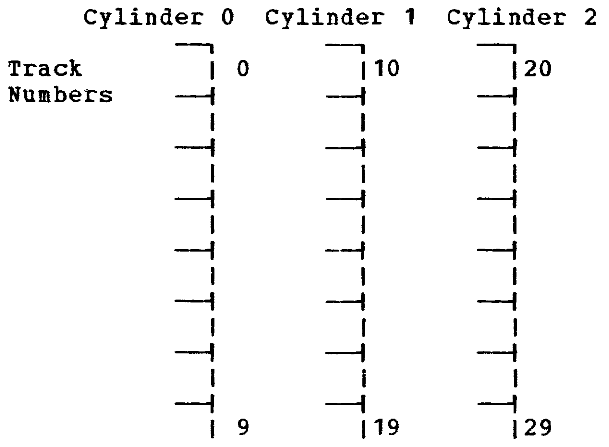
Figure 32. Creating a Direct File with Relative Track Addressing Using Method B (Part 4 of 4)

ACTUAL TRACK ADDRESSING CONSIDERATIONS FOR SPECIFIC DEVICES

Randomizing for the 2311 Disk Drive

When randomizing for the 2311 Disk Drive, it is possible to circumvent the discontinous binary address by coding the randomizing formula in decimal arithmetic and then converting the results to binary. This can be done by setting aside a decimal field with the low-order byte reserved for the head number, and the high-order bytes reserved for the cylinder number. A mathematical overflow from the head number will now increment the cylinder number and produce a valid address. The low-order byte should then be converted to binary and stored in the HH field, and the high-order bytes converted to binary and stored in the CC field of the actual key field.

Randomizing to the 2311 Disk Drive should present no significant problems if the programmer using direct organization is completely aware that the cylinder and head number give him a unique track number. To illustrate, the 2311 could be thought of as consisting of tracks numbered as follows:

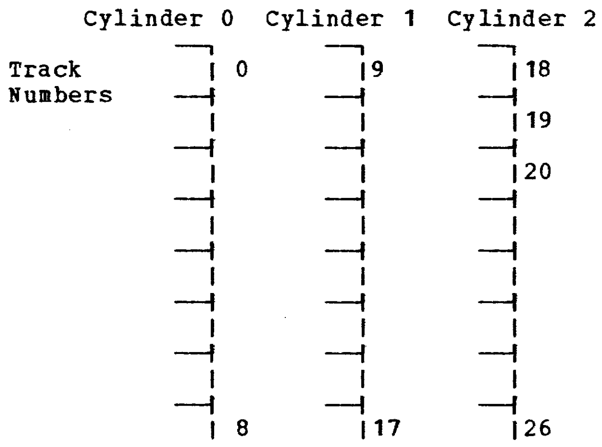


If the randomizing formula resulted in an address of cylinder 001, head 9:

Cylinder Number	Head Number
001	9

this would be a reference to track 19. This fact allows the programmer to ignore the discontinuous cylinder and head number. If his formula resulted in an address of 0020, this would result in accessing cylinder 2, head 0, the location of track 20.

The programmer can make another use of this decimal track address. He may wish to reserve the last track of each cylinder for synonyms. If this is the case, he is in effect redefining the cylinder to consist of nine tracks rather than ten tracks. The 2311 cylinder could then be thought of as consisting of track numbers, as follows:



If the programmer randomizes to relative track number 20, he can access it by dividing the track address by the number of tracks (9) in a cylinder. The quotient now becomes the cylinder number, and the remainder becomes the head number.

```
       2 = cylinder number
    ____
 9 )0020
      18
     ---
       2 = head number
```

To simplify randomizing, an algorithm must be developed to generate a decimal track address. This track address can then be converted to a binary cylinder number and head number. In addition, tracks can be reserved by dividing the track address by the number of tracks in a cylinder. The same concepts will hold true for devices such as the 2314, 3330, or 3340. For example, an algorithm can be developed using 20 tracks per cylinder and dividing by the closest prime number less than 20.

Randomizing for the 2321 Data Cell

The track reference field for the 2321 Data Cell is composed of the following discontinuous binary address:

		cell	sub cell	strip	cyl.	head	record
M	B	B	C	C	H	H	R
		0-9	0-19	0-9	0-4	0-19	

At first glance, this presents an almost impossible randomizing task; but since each strip includes 100 tracks that are accessible through cylinder and head number, the 2321 Data Cell can be considered to consist of consecutively numbered tracks.

Tracks	Strip
0 → 99	0
100 ↔ 199	1
900 ↔ 999	9
1000 ↔ 1099	10
1900 ↔ 1999	19
10000 ↔ 10099	100
19900 ↔ 19999	199
199900 ↔ 199999	1999

It can be seen that relative track 20 is located on cylinder 1, head 0 of some particular strip. Its address can be calculated by dividing by 20.

```
      1 = cylinder number
   ___
20 ) 20
     20
    ---
      0 = head number
```

Thus, relative track number 120 will be located on strip 1, cylinder 1, head 0 of some subcell. Note that the strip number is given by the hundreds digit, and the cylinder and head number are derived by dividing the two low-order digits by 20.

The same relationship holds true for relative track number 900. It is located on strip 9, cylinder 0, track 0. Again, the hundreds digit gives the strip number, and dividing the two low-order digits by 20 results in a quotient and remainder of zero.

This relationship holds true through a relative track number of 19999, which is the number of tracks that can be contained on one cell of a data cell array. By applying the foregoing rules, an address of subcell 19, strip 9, cylinder 4, head 19 is derived.

Thus, by randomizing to a 5-digit decimal track number, the programmer will be able to access the 20,000 tracks (40,000,000 characters) contained in a cell.

The thousands digits would represent the subcell number, the hundreds digit the strip number, and the quotient and remainder of the two low-order digits divided by 20 would represent the cylinder and head number. Each one of these resulting decimal digits would then be converted to binary and placed in the appropriate location in the track reference field.

There is a total of 200,000 tracks per data cell array. To derive valid addresses that cross cell boundaries, the programmer should randomize to a 6-digit decimal track address. The highest address possible should be 199,999. To convert this to a data cell address, similar rules apply. In this case, the programmer must divide the three high-order digits by 20:

```
        9 = cell
    ____
20 ) 199
     180
    ----
      19 = subcell
```

The quotient becomes the cell number and the remainder becomes the subcell number. The hundreds digit is still the strip number, and the cylinder and head number can be derived as previously illustrated. The resulting address is 0091994190 and would appear in the first eight bytes of the actual key field as follows:

		cell	sub cell	strip	cyl.	head	
M	B	B	C	C	H	H	R
0	0	9	19	9	4	19	0

Randomizing to the data cell can be accomplished by developing an algorithm to generate decimal track addresses. The use of the foregoing rules makes it possible to convert these generated track addresses to the appropriate discontinuous binary address.

INDEXED ORGANIZATION (DTFIS)

An indexed file is a sequential file with indexes that permit rapid access to individual records as well as rapid sequential processing. Error recovery from a DTFIS file is described in detail in the chapter "Advanced Processing Capabilities." An indexed file has three distinct areas: a prime area, indexes, and an overflow area. Each area is described in detail below.

PRIME AREA

When the file is first created, or when it is subsequently reorganized, records are written in the prime area. Until the prime area is full, additions to the file may also be written there. The prime area may span multiple volumes. Note that the last track of the prime area may not be used by the COBOL programmer.

The records in the prime area must be formatted with keys, and must be positioned in key sequence. The records may be blocked or unblocked. If records are blocked, each logical record within the block contains its key, and the key area for the block contains the key of the highest record in the block. The Disk Operating System Virtual Storage permits fixed-length records only. Figure 33 shows the formats of blocked and unblocked records on a track.

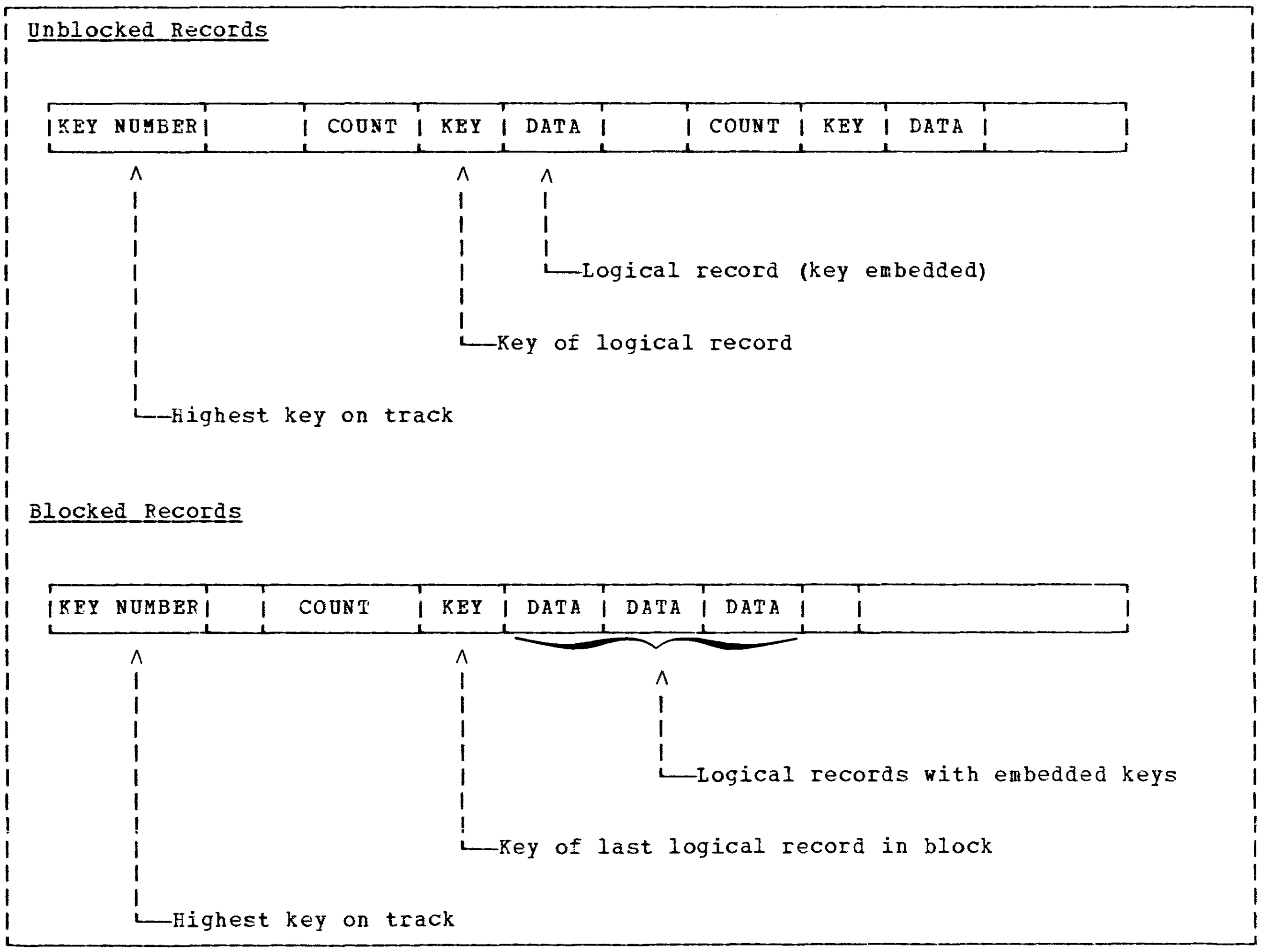


Figure 33. Formats of Blocked and Unblocked Records

INDEXES

There are three possible levels of indexes for a file with indexed organization: a track index, a cylinder index, and a master index. They are created and written by the system when the file is created or reorganized.

Track Index

This is the lowest level of index and is always present. There is one track index for each cylinder in the prime area. It is always written on the first track of the cylinder that it indexes.

The track index contains a pair of entries for each prime data track in the cylinder: a normal entry and an overflow entry. The normal entry contains the home address of the prime track and the key of the highest record on the track. The overflow entry contains the highest key associated with that track and the address of the lowest record in the overflow area. If no overflow entry has yet been made, the address of the lowest record in the overflow area is the dummy entry X'FF'.

Cylinder Index

The cylinder index is a higher level of index and is always present. Its entries point to track indexes. There is one cylinder index for the file. It is written on the device specified in the APPLY CYL-INDEX clause. If this clause is not specified, the cylinder index is written on the same device as the prime area.

Master Index

The master index is the highest level index and is optional. It is used when the cylinder index is so long that searching it is very time consuming. It is suggested that a master index be requested when the cylinder index occupies more than four tracks. (A master index consists of one entry for each track of the cylinder index.)

The DOS/VS System permits one level of master index for the file and requires that it be written immediately before the cylinder index. If a master index is desired, the APPLY MASTER-INDEX clause must be specified in the source program. When this clause is specified, the cylinder index is placed on the same device as the master index.

Note: The indexes are terminated by a dummy entry containing a key composed of all ones (bits). To avoid any possibility of errors, the user should not specify a key of all ones (HIGH VALUES) for any of his records.

OVERFLOW AREA

There are two types of overflow areas: a cylinder overflow area and an independent overflow area. Either or both may be specified for an indexed file. Records are written in the overflow area(s) as additions are made to the file.

Cylinder Overflow Area

A certain number of whole tracks are reserved in each cylinder for overflow records from the prime tracks in that cylinder. The programmer may specify the number of tracks to be reserved by means of the APPLY CYL-OVERFLOW clause. If he specifies 0 as the number of tracks in this clause, no cylinder overflow area is reserved. If the clause is omitted, 20% of each cylinder is reserved for overflow. For the 3330, three tracks of each cylinder will be reserved for overflow. For the 3340, two tracks of each cylinder will be reserved for overflow. When an ISAM file has been created with the APPLY CYL-OVERFLOW clause all FD's, which use the same file, must specify the same number of cylinder overflow tracks.

Independent Overflow Area

Overflow records from anywhere in the prime area are placed in a certain number of cylinders reserved soley for this purpose. The size and location of the independent overflow area can be specified if the programmer includes the proper job control EXTENT cards. The area must, however, be on the same mass storage device type as the prime area.

A suggested approach is to have cylinder overflow areas large enough to contain the average number of overflow records caused by additions and an independent overflow area to be used as the cylinder overflow areas are filled.

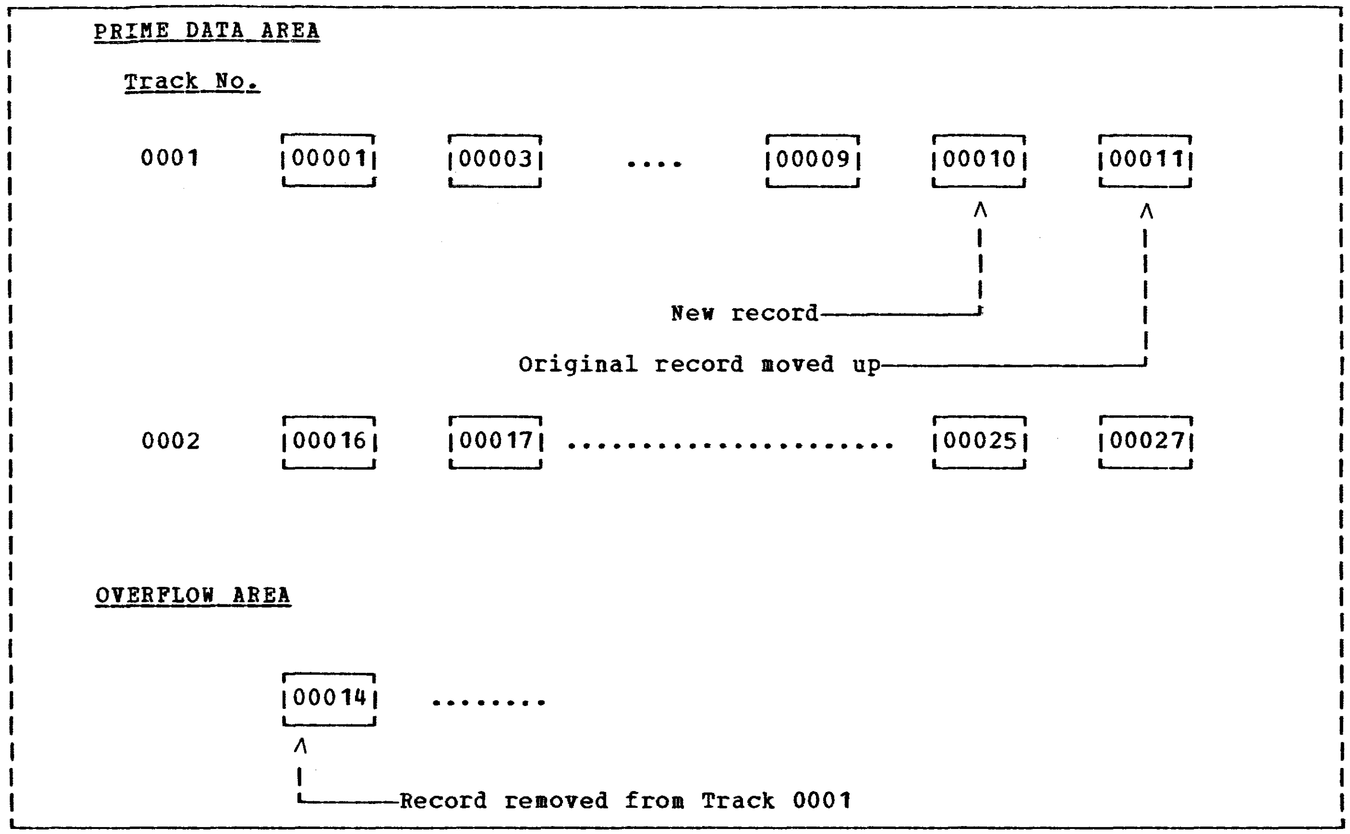


Figure 34. Adding a Record to a Prime Track

Adding Records to an Indexed File

A new record added to an indexed file is placed into a location on a track in the prime area determined by the value of its key field. If records in the file were placed in precise physical sequence, the addition of a new record would require the shifting of all records with keys higher than that of the one inserted. However, indexed organization allows a record to be inserted into its proper position on a track, with the shifting of only the records on that track. Any records for which there is no space on that track are then placed in an overflow area, and become overflow records. Overflow records are always fixed-length, unblocked records, formatted with keys.

As records are added to the overflow area, they are no longer in key sequence. The system ensures, however, that they are always in logical sequence.

Figure 34 illustrates the addition of a record to a prime track.

The new record (00010) is written in its proper sequential location on the prime track. The rest of its prime records are moved up one location. The bumped record (00014) is written in the first available location in the overflow area. The record is placed in the cylinder overflow area for that cylinder, if a cylinder overflow area exists and if there is space in it; otherwise, the record is placed in the independent overflow area. The first addition to a track is always handled in this manner. Any record that is higher than the original highest record on the preceding track, but lower than the original highest record on this track, is written on the prime track. Record 00015, for example, would be written as the first record on track 0002, and record 00027 would be bumped into the overflow area.

Subsequent additions are written either on the prime track where they belong or as part of the overflow chain from that track. If the addition belongs between the last prime record on a track and a previous overflow from that track (as is the case with record 00013), it is written in the first available location in the overflow area on an empty track, or on a track whose first record has a numerically lower key.

If the addition belongs on a prime track (as would be the case with record 00005), it is written in its proper sequential location on the prime track. The bumped record (record 00011) is written in the overflow area.

A record with a key higher than the current highest key in the file is placed on the last prime track containing data records. If that track is full, the record is placed in the overflow area.

ACCESSING AN INDEXED FILE (DTFIS)

An indexed file may be accessed both sequentially and randomly.

ACCESSING AN INDEXED FILE SEQUENTIALLY: An indexed file may only be created sequentially. It can also be read and updated in the sequential access mode. The following specifications may be made in the source program.

ENVIRONMENT DIVISION

Required clauses:

```
   SELECT [OPTIONAL] file-name

                      (2311)
                      (2314)
   ASSIGN TO SYSnnn-DA-(2321)- I
                      (2319)
                      (3330)
                      (3340)
   RECORD KEY Clause
   NOMINAL KEY Clause (when reading, if the
     START statement is used)
```

Optional clauses:

```
   FILE-LIMIT Clause
   ACCESS MODE IS SEQUENTIAL
   PROCESSING MODE IS SEQUENTIAL
   RERUN Clause
   SAME Clause
   APPLY WRITE-VERIFY Clause (create and
     update)
   APPLY CYL-OVERFLOW Clause (create)

         {MASTER-INDEX}
   APPLY {            }Clause
         {CYL-INDEX   }

   RESERVE Clause
```

Invalid clauses:

```
   ACCESS MODE IS RANDOM
   ACTUAL KEY Clause
   TRACK-AREA Clause
   MULTIPLE FILE TAPE Clause
   APPLY WRITE-ONLY Clause
   APPLY EXTENDED-SEARCH Clause
   APPLY CORE-INDEX Clause
```

ACCESSING AN INDEXED FILE RANDOMLY: A randomly-accessed indexed file may be read, updated, or added to. The following specifications may be made in the source program:

ENVIRONMENT DIVISION

Required clauses:

```
   SELECT [OPTIONAL] file-name

                         (2311)
                         (2314)
   ASSIGN TO SYSnnn-DA-  (2321) -I
                         (2319)
                         (3330)
                         (3340)
   ACCESS IS RANDOM
   NOMINAL KEY Clause
   RECORD KEY Clause
```

Optional clauses:

```
   FILE LIMIT Clause
   PROCESSING MODE IS SEQUENTIAL
   TRACK-AREA Clause
   RERUN Clause
   SAME Clause
   APPLY WRITE VERIFY Clause
   APPLY CYL-OVERFLOW Clause
   APPLY CORE-INDEX Clause

         {MASTER-INDEX}
   APPLY {            }    Clause
         {CYL-INDEX   }
```

Invalid clauses:

```
   RESERVE Clause
   ACCESS MODE IS SEQUENTIAL
   ACTUAL KEY Clause
   MULTIPLE FILE TAPE Clause
   APPLY EXTENDED-SEARCH Clause
```

Key Clauses

When creating an indexed file, the only key clause required is the RECORD KEY clause. The data-name specified in this clause is the name of the field within the record that contains the key. Keys must be in ascending numerical order when creating an indexed file.

If a START statement is used when retrieving an indexed file sequentially, the NOMINAL KEY clause is required.

When accessing an indexed file randomly, both the NOMINAL KEY and RECORD KEY clauses are required. When reading the file, the data-name specified in the NOMINAL KEY clause is the key of the record which is being retrieved. The data-name specified in the RECORD KEY clause is the name of the field within the record that contains this key.

When adding records to an indexed file, the data-name specified in the NOMINAL KEY clause is the key for the record being written and is used to determine its physical location. The data-name specified in the RECORD KEY clause specifies the field in the record that contains the key.

Note: If an INVALID KEY exit is taken on a START statement, the key value in the NOMINAL KEY data-name should be corrected and another START statement issued to ensure correct retrieval of blocked records.

Improving Efficiency

When processing an indexed file, the following source language Environment Division clauses may be used to improve efficiency:

TRACK-AREA Clause
APPLY CORE-INDEX Clause

For additional details, see the publication *IBM DOS Full American National Standard COBOL*.

The DOS/VS Compiler supports 3540 Diskette unit file management. This device is quite different from standard direct access devices as it does not access data randomly. The medium used for reading and writing is a diskette which can be easily mailed from one location to another.

Data can be recorded on the 3540 diskette in two ways:

1. Keypunching on the diskette via the 3740 processing device.

2. Writing sequential data sets on the diskette via the 3540 Diskette unit attached to a System/370.

DOS/VS COBOL processing applies only to the processing of data on the diskette by the 3540 Diskette unit.

For the use of system files on diskette, see DOS/VS System Management Guide.

FILE PROCESSING

File processing for the 3540 is sequential only. Only fixed-length physical records can reside on the diskette. Logical blocking of records is an available function and will be discussed in the section entitled "Cobol Language Considerations."

The system interfaces with the COBOL object module through DTFDU, (generated as part of the object module), and DUMOD logic modules (used to perform actual I-O processing). The generated DTFDU will correspond to a DTFDU generated by the DTFDU macro (described in DOS/VS Supervisor and I-O Macros) with the exceptions specified later in this section.

The physical considerations of the 3540 diskette include:

- The diskette is divided into character sectors with each sector containing 128 characters.

- Each record may occupy no more than one sector, and may be from 1 to 128 characters long.

- Each record in a file must be the same size.

- Blocking factors can be only 1, 2, 13, or 26 records.

Files may be extended to additional diskettes if one diskette is too small. This is done automatically by LIOCS if DLBL and EXTENT cards are provided for additional processing. There is no user program control to force end of volume for this device.

File labels exist on the 3540 Diskette for each file, but no user control or processing of these labels is provided by the DOS/VS system. Label management will be handled strictly by LIOCS. The user will only have to provide the name for the file in the DLBL control card.

COBOL LANGUAGE CONSIDERATIONS

ENVIRONMENT DIVISION

The following format of the SELECT statement applies to the 3540:

Required clauses:

```
SELECT [OPTIONAL] file-name

ASSIGN TO SYSnnn-{UT}-3540-S[-name]
                 {DA}
```

Sort work files may not be assigned to the 3540. A 3540 may not be a checkpoint device.

Optional clauses:

```
RESERVE clause
ACCESS MODE IS SEQUENTIAL Clause
PROCESSING MODE IS SEQUENTIAL clause
RERUN ON system-name EVERY integer
  RECORDS OF file-name
 (System-name cannot specify 3540;
  file-name can refer to 3540 file;
  checkpoint records cannot be taken on
  a diskette, but a diskette can be used
  to control when checkpoints are
  taken.)
SAME clause
FILE LIMIT clause
```

Invalid Clauses:

```
APPLY WRITE-ONLY clause (only
  fixed-length records allowed)
APPLY WRITE-VERIFY clause (function not
  supported)
ACCESS MODE IS RANDOM clause
ACTUAL KEY clause
```

NOMINAL KEY clause
RECORD KEY clause
TRACK-AREA clause
MULTIPLE FILE TAPE clause
RERUN clause (see restrictions above)
APPLY EXTENDED-SEARCH clause
APPLY CYL-OVERFLOW Clause

APPLY $\left\{\begin{array}{l}\text{MASTER-INDEX}\\ \text{CYL-INDEX}\end{array}\right\}$ clause

APPLY CORE-INDEX clause

DATA DIVISION

The following restrictions apply to the FD and record description for a 3540 file:

- Recording mode must be F.
- Label records must be standard.
- RECORD CONTAINS clause cannot specify more than 128 characters, or "integer-1 to integer-2" CHARACTERS.
- The BLOCK CONTAINS clause must specify the RECORDS option only. Blocking is permitted for the most efficient usage of the 3540. If this clause is specified, only 1, 2, 13, or 26, will be accepted as the blocking factor. Any other number will cause a diagnostic.
- In the record description, a maximum of 128 characters will be allowed for a 3540 file.
- The record description for a 3540 file must not include any items with the OCCURS DEPENDING ON clause, as variable records are not allowed.

Procedure Division -- Special Considerations

- OPEN Statement. 3540 files may be opened for input or output only. Since updating is not permitted for a 3540 file, OPEN I-O is not allowed.
- Only one 3540 file per diskette may be open simultaneously.
- The REVERSED and NO REWIND options of the OPEN statement are not valid for a 3540 file.
- WRITE Statement. The INVALID KEY option may not be used for a 3540 file. If the end of the diskette is reached and additional diskette information has not been supplied via additional EXTENT control cards, the operator will be queried to either supply an EXTENT through the console or cancel the job.
- Standard errors can be handled in a USE AFTER STANDARD ERROR Declarative. Two types of errors will cause control to return to an error declarative for 3540 files:

  1. Data check
  2. Equipment check

  If the GIVING option is specified, byte 1 will indicate a data check, and byte 2 will indicate an equipment check.

  In either case, the error procedure is used to continue processing or to close the file. If processing continues and the file is blocked, the remaining records in the block after the record causing the error may be lost when the next READ or WRITE statement is executed.

  If no error declarative is specified, a message will be issued describing the type of error, and the job will be canceled.
- CLOSE Statement. When a CLOSE statement is executed for a 3540 file, the present diskette will be fed out into the output hopper. CLOSE UNIT may not be used as no forced end-of-volume support is included for the 3540 Diskette unit. CLOSE NO REWIND may not be used. The LOCK option will be supported for 3540 files.

DTFDU

The compiler will generate DTFDU with the following defaults:

1. No write protection
2. Feed = yes
3. Volume sequencing will be checked.
4. No read/write security.

Job Control Requirements

Normal job control DLBL and EXTENT statements for the 3540 are shown below.

DLBL Statement

The format of the DLBL statement is:

```
// DLBL filename,['file-ID'],[date],[code]
```

filename -- is a unique filename of 3 to 7 characters identical to the symbolic name of the DTF that identifies the file. Supported in the same way as for current devices. This corresponds to the "name" field of system-name in the SELECT statement if specified, or to SYSnnn in the system-name.

'file-ID' -- only the first 8 characters will be used. Supported in the same manner as for current devices.

date -- provides the expiration date for the file. Supported in the same way as for current devices.

code -- is a field indicating the type of file label. DU for diskette unit is supported. It is supported in the same way as for current devices.

EXTENT Statement

The format of the EXTENT statement is:

```
// EXTENT [symbolic-unit],
          [serial-number],[1]
```

symbolic unit -- indicates the symbolic unit (SYSxxx) of the volume for which the extent is effective. It is supported in the same way as for current devices.

serial number -- indicates the volume serial number of the volume for which this extent is effective. It is supported in the same way as for other devices. The serial number is optional. If omitted, the volume that is mounted is assumed to be the correct volume.

type -- indicates the type of extent. A '1' indicates 'data area.' No other types are supported.

3540 File

The following DLBL and EXTENT statements describe a file that resides on a 3540 diskette.

```
// DLBL MASTER,,75/001,DU
// EXTENT SYS015,111111,1
```

In the following example, the program CREATES creates a diskette (DU) file named SALES that is to be retained until the end of 1975. The file comprises up to three diskettes. The diskettes have the volume serial numbers 111111, 111112, and 111113, and are mounted on the drive assigned to the symbolic device name SYS005.

```
// JOB EXAMPLE
// ASSGN SYS005,X'060'
// DLBL SALES,'ANNUAL',75/365,DU
// EXTENT SYS005,111111,1
// EXTENT SYS005,111112,1
// EXTENT SYS005,111113,1
// EXEC CREATE
/&
```

The COBOL statements which correspond to this are:

```
SELECT SALES-FILE ASSIGN
    TO SYS005-DA-3540-S-SALES.
        .
        .
        .
FD SALES-FILE
RECORDING MODE IS F
LABEL RECORDS ARE STANDARD
RECORD CONTAINS 80 CHARACTERS.

01  DISKETTE-RECORD.
    02
        .
        .
        .
```

VSAM is a new access method for direct or sequential processing of fixed and variable length records on direct-access devices. It has more functions, generally better performance, better data integrity and security, improved data organization, and is easier to use and control than the DOS/VS DAM and ISAM access methods.

VSAM files can be processed only by the VSAM file processing technique. The programmer can convert SAM and ISAM files to VSAM files by using the method described in the section entitled "Converting Non-VSAM Files to VSAM Files." The following topics related to VSAM are discussed in this chapter:

VSAM File processing
Access Method Services
Error Handling

File Organization

The records in a VSAM file can be organized either in logical sequence by a key field (key-sequence) or in the physical sequence in which they are written on the file (entry-sequence).

A key-sequenced file has an index, like ISAM; the records in a key-sequenced file can be accessed by key, either randomly or sequentially. An entry-sequenced file does not have an index, and records can be accessed sequentially only.

Key-Sequenced Files

Like ISAM files, key-sequenced files are ordered according to a user-defined key field in each record. That is, they are ordered according to the collating sequence of the key field in each record. Each record has a unique value in the key field, such as employee number or invoice number. VSAM uses the key associated with each record to insert a new record in the file or to retrieve a record from the file. The order of access can be random or sequential. Key-sequenced files, however, can generally be processed faster than ISAM files because VSAM has a more efficient index and does not use chained record overflow.

When a key-sequenced file is created, certain portions can be left empty, that is, free space can be distributed throughout the file. This free space is used when inserting new records or lengthening existing records. This eliminates the need for overflow chains and overflow areas; it also minimizes data movement. Thus performance does not degrade substantially as records are added and the file does not have to be reorganized as often as an ISAM file. VSAM reclaims space when a record is deleted or shortened, and the space released becomes free space.

The index of a key-sequenced VSAM file is more efficient than an ISAM index because it generally requires less direct-access space and less updating of index entries. Space is saved in three ways: by eliminating redundant key information (key compression), by having fewer keys in the index than there are records in the file (non-dense index), and by blocking index records. A shorter index requires less time to search and update. Updating is infrequent, because index entries are not usually modified when records are added to or deleted from the file.

A key-sequenced file is defined in COBOL by specifying:

```
SELECT file-name ASSIGN TO
   SYSnnn[-class][-device][-name]
   ORGANIZATION IS INDEXED....
   RECORD KEY IS...
```

Entry-Sequenced Files

Records are stored in entry-sequenced files in the order they are presented for inclusion on the file (that is, their entry-sequence), and without respect to the contents of the records. No keys are recognized and, consequently, no indexes are built. The order of records is fixed; they are not moved. Thus, free space is not distributed throughout the file and new records are placed at the end. Records cannot be shortened, deleted, or lengthened. Since there is no index, the user must access the file sequentially (in the order the records were written).

An entry-sequenced file is defined in COBOL by specifying:

```
SELECT file-name ASSIGN TO
   SYSnnn[-class][-device]-AS[-name]
   ORGANIZATION IS SEQUENTIAL....
```

Data Organization

The data organization of ISAM is based on the physical units of disk cylinder and disk track, while the data organization of VSAM is based on logical units called control intervals and control areas. A control interval is the unit of direct-access storage that is transferred to and from virtual storage. It can contain one or more records in one or more blocks. Each entry in the lowest index level of a key-sequenced VSAM file points to a control interval. Free space in a key-sequenced file is distributed in terms of the percent of total space. A percentage of each control interval can be free space and some control intervals can be entirely free space. Indexes are also organized in control intervals. Each contains a single index record which can have many index entries. A control area is a group of control intervals. VSAM data organization provides for device independence by reducing the programmer's concern about the physical characteristics of the data and the index. Figure 35 illustrates VSAM data and index structure.

Data Access

Key sequenced files can be accessed either sequentially, or directly by key. The key used can be either the full key or a generic key (any front part of the full key).

The COBOL user can retrieve, add, update or delete records from a VSAM file by means of the READ, WRITE, REWRITE and DELETE verbs. Also, by means of the START verb he can position himself to any record in the file and begin sequential retrieval from that record.

VSAM Catalog

VSAM keeps central control over the creation, access, and deletion of files and over the management of direct-access storage space allocated to those files. This is done by keeping information on file and space characteristics in one place, the VSAM catalog. The catalog, which is unique to VSAM, makes it easier to (1) keep track of files and available direct-access space, (2) write job control statements to create and process VSAM files, and (3) move VSAM files to other DOS/VS systems or to OS/VS systems. There can be more than one VSAM catalog. However, only one catalog at a time can be connected to the system. Each catalog can keep track of VSAM files on many volumes; it is not necessary to mount a volume to determine whether or not it has space available for a VSAM file.

Figure 35 shows the structure of the data and index in a VSAM file. It does not represent accurate proportions in terms of the number of records in a control interval, etc.

In the example, if the user wanted to add a record whose record key was 1048, it logically belongs between records 1024 and 1068. This is where VSAM would insert the record physically. The record with key 1068 would be moved over in the control interval taking up free space, to make room for the new record. This movement of records is done in core before any writing takes place.

This example illustrates several points:

1) New records are physically inserted where they logically belong with only local record movement required. Thus, new records are retrieved in the same fashion as are old records.

2) Since the index pointers are non-dense (one for each control interval rather than one for each record), the insertion of the record requires no change to the index.

3) Record movement for insertion, deletion and updating takes place in core, before any I/O takes place, thus improving data integrity.

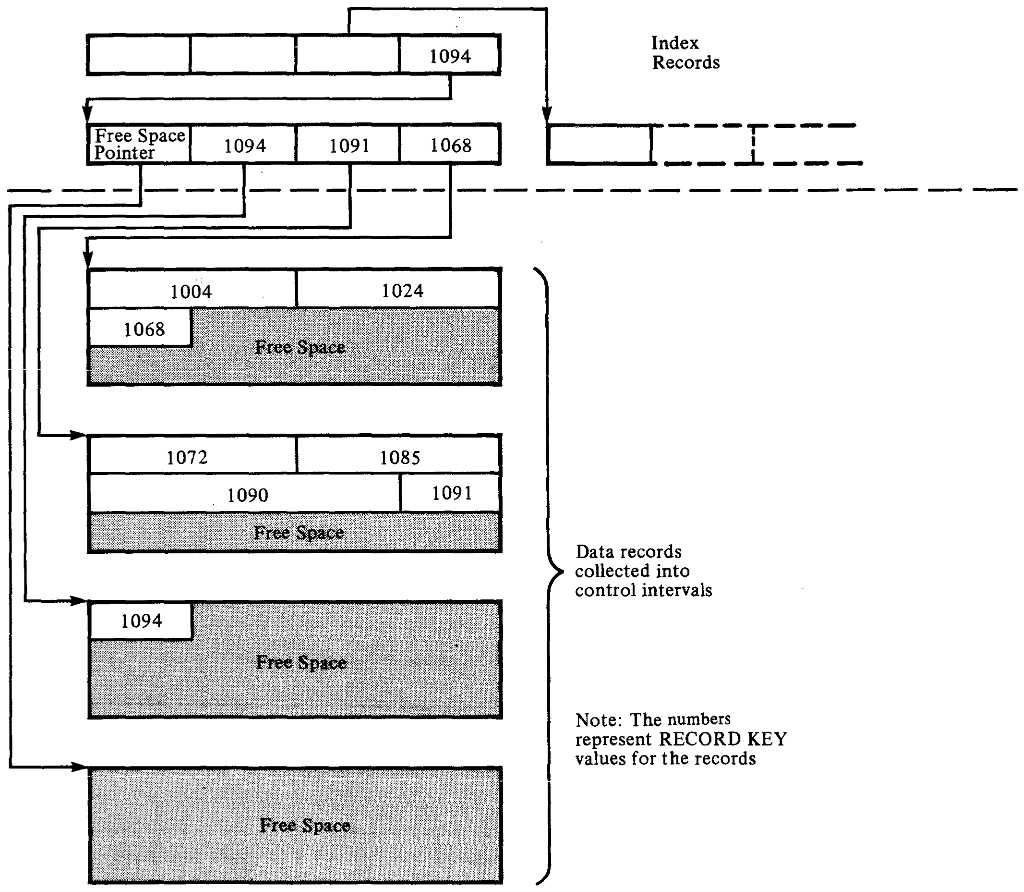


Figure 35. VSAM Data Organization

File and Volume Portability

A significant feature of VSAM is that files can be moved from one DOS/VS system to another or to an OS/VS system. This is possible because VSAM data format is identical under both DOS/VS and OS/VS.

Service Programs

VSAM has an extensive service program package, called Access Method Services, which can be used to:

- Define, print, copy , or reorganize VSAM files.
- Add, alter, delete, or print catalog entries.
- Convert ISAM and SAM files to VSAM files.
- Export and import files from one system to another.

Device Support

VSAM files can be written on 2314, 3330, 3340, 3350, and fixed block devices.

Security

Through COBOL, access to the file can be restricted by use of the PASSWORD clause in the SELECT statement.

Error Processing

VSAM provides exits to a user-supplied routine to handle I/O and/or logical errors or exeception conditions. This is done in COBOL via the USE AFTER STANDARD ERROR declarative and the INVALID KEY and AT END clauses. A STATUS KEY may be specified, and the details of the condition determined.

VSAM Messages

Like other access methods, VSAM issues messages to the operator, if for example, the incorrect volume is mounted, etc. These messages are described in DOS/VS Messages. VSAM Access Method Services also issues messages to the programmer which are documented in DOS/VS Access Method Services. COBOL issues VSAM messages to the operator and/or programmer. These are listed in "Appendix I: Diagnostic Messages."

For more detail on VSAM, refer to DOS/VS Access Method Services.

ACCESS METHOD SERVICES

Access Method Services is a utility program. A number of user-entered commands, either modal or functional, initiate the Access Method Services programs. The functional commands invoke the desired Access Method Services function while the modal commands control the sequence of execution of the functional commands. In this chapter, only certain commands and parameters are discussed. For complete details on the use of commands see DOS/VS Utilities Access Method Services.

Functional Commands

There are nine functional commands: DEFINE, ALTER, DELETE, LISTCAT, REPRO, PRINT, IMPORT, EXPORT, and VERIFY. The commands DEFINE, ALTER, and DELETE are used to create, modify, and remove VSAM catalogs and files. LISTCAT is used to list the contents of a VSAM catalog. The REPRO and PRINT commands reproduce files either as new files or as printed output. The IMPORT and EXPORT commands provide for transfers of files from one system to another. The VERIFY command provides a file recovery service for VSAM files by ensuring that the end of the file indicated in the catalog is the same as the actual file end.

The DEFINE Command

All VSAM files must be cataloged in a VSAM catalog. This catalog must be defined and allocated by Access Method Services. This is the first step which must be taken by a user who plans to use VSAM.

The DEFINE command is used to define a VSAM object. In VSAM terminology, an object is either a VSAM catalog, a VSAM data space, or a VSAM file.

VSAM files must be cataloged in a VSAM catalog. Non-VSAM files may also be cataloged in a VSAM catalog. All VSAM files are introduced to the system through the DEFINE command.

There are two steps in the creation of an object: defining the object in the catalog, and generating the contents of that object. The DEFINE command simply makes an entry in the catalog, it does not generate any content.

Specification of the DEFINE Command

```
|                 Format                  |
|-----------------------------------------|
| DEFINE  object  parameters              |
```

The definable objects are:

- MASTERCATALOG -- specifies that the VSAM master catalog is to be defined.
- SPACE -- specifies that a VSAM data space is to be defined.
- CLUSTER -- specifies that a file is to be defined.

For each file there is an associated valid parameter list.

Defining a VSAM Master Catalog: DEFINE MASTERCATALOG

The DEFINE MASTERCATALOG command must be used to set up the master catalog. It is the first Access Method Services command used since without a master catalog other objects cannot be defined. Defining a master catalog is somewhat different from defining a file. When the user defines a file he need not necessarily allocate space as part of the define operation. However, the process of defining catalog always involves the allocation of space for that catalog. Entries for both the master catalog itself and the volume containing the data space automatically created are placed in the master catalog.

The following is an example of defining a VSAM master catalog.

```
// JOB    DEFINE A VSAM CATALOG
// DLBL   IJSYSCT,'VSAMCAT',,VSAM
// EXTENT SYSCAT,321940,1,,100,250
// EXEC   IDCAMS,SIZE=26K
   DEFINE MASTERCATALOG(NAME(VSAMCAT)   -
          VOLUME (321940) TRACKS (250)  -
          FILE(IJSYSCT) UPDATEPW(SECRET)-
          READPW(NOSECRET))
/*
/&
```

Figure 36. Defining a VSAM Master Catalog

The DLBL statement must be used to specify the filename and the code which identifies VSAM. The filename must be specified as IJSYSCT.

The logical unit in the EXTENT statement must be SYSCAT. The user must decide which volumes and which extents will contain the catalog. Note that the VOLUMES parameter and the space allocation parameter (CYLINDERS, TRACKS, or RECORDS) must be included in the DEFINE command, and must agree with the information in the EXTENT statement. If the CYLINDERS parameter is used, each extent must begin on a cylinder boundary.

The following parameters were used in the above example:

NAME (VSAMCAT)
: The name of the VSAM master catalog is VSAMCAT. All future references to the catalog are made using this name.

VOLUME (321940)
: The volume serial number on which the catalog is to reside is 321940.

TRACKS (250)
: The number of tracks allocated to the catalog is 250. This must agree with the information on the EXTENT card.

: Note that every key-sequenced file requires three catalog entries: one each for the cluster, data component, and index component. Every entry-sequenced file requires two catalog entries: one for the cluster and one for the data component.

FILE (IJSYSCT)
: This parameter identifies the filename of the DLBL statement that specifies the device and volume for allocation. The filename must be specified as IJSYSCT.

UPDATEPW (SECRET)
The update level password is SECRET. This is an optional parameter. However, if any file which is cataloged in the VSAM catalog is to be password protected, the catalog itself must also be password protected.

READPW (NOSECRET)
The read level password is NOSECRET. This is an optional parameter. If specified, all reading of the catalog requires this password.

There are 4 levels of password protection for a VSAM catalog or file. They are: master level (this is the highest level of protection), the CI level (this is a special case and should not be used with COBOL), the update level and the read level (the lowest level of protection).

If password protection is not specified at a higher level, but is specified at a lower level, then the lower level password becomes the password for the higher levels which are not specified. If password protection is not specified for the lowest level (read level) then there is no password protection for that lowest level or for the higher levels which are not specified.

So in the example, SECRET is the master level password as well as the update level password, since the master level password was not specified.

The update level password of the catalog is required in order to change the content of the catalog, for example to DEFINE or DELETE a file in that catalog.

Defining a VSAM Data Space: DEFINE SPACE

VSAM data space is space which is owned and managed by VSAM. When space on a volume is defined in a VSAM catalog then that volume is said to be owned by that VSAM catalog. This means that no other VSAM catalog can own space on that volume. It does not mean that there can be no non-VSAM space on the volume.

VSAM data space can contain the records for one file or for many files, but all the files occupying a VSAM data space must be cataloged in the same VSAM catalog as is the space.

Since the process of defining VSAM data space necessarily requires the allocation of space, JCL is required for extent information.

Figure 37 is an example of defining a VSAM data space:

```
// JOB       DEFINE A VSAM DATA SPACE
// ASSGN     SYS001,X'130'
// DLBL      VFILENM,,,VSAM
// EXTENT    SYS001,321942,1,,800,400
// EXEC      IDCAMS,SIZE=26K
   DEFINE    SPACE (FILE(VFILENM)       -
             TRACKS (400)      -
             VOLUMES(321942))       -
             CATALOG(VSAMCAT/SECRET)
/*
/&
```

Figure 37. Defining a VSAM Data Space

The DLBL statement must be used to specify the filename and the code which identifies VSAM files. The filename (VFILENM) is the same as the FILE parameter and connects the job control statements to the DEFINE command. The EXTENT statement must be used to specify the symbolic unit name, the volume serial number, and the space parameters. The VOLUMES parameter and the space allocation parameter (CYLINDERS, TRACKS, or RECORDS) must be included in the DEFINE command, and must agree with the information in the EXTENT statements. If the CYLINDERS parameter is used, each extent must begin on a cylinder boundary.

The following parameters were used in Figure 37.

FILE (VFILENM)
This required parameter identifies the filename of a DLBL statement that specifies the devices and volumes to be used for space allocation.

TRACKS (40D)
This parameter specifies the amount of space to be allocated in terms of tracks. The number used to specify the tracks to be allocated to the data space must agree with the information in the extent statements.

VOLUMES (321942)
This required parameter specifies the volumes to contain the data spaces. If more than one volume is specified, each volume will contain a data space of the same size. Note that the VOLUMES parameter must agree with the information in the EXTENT statements. The volume serial number of the volume(s) containing the data space(s) is substituted for volser.

CATALOG (VSAMCAT/SECRET)
This is a required parameter if the master catalog is password protected. It specifies the name of the catalog which is to own the space, and the update password for that catalog.

Defining a VSAM File: DEFINE CLUSTER

DEFINE CLUSTER is used to define all attributes of all VSAM files and to catalog the files in a VSAM catalog.

Note: This command cannot be used to add records to the VSAM file.

VSAM files can be sub-allocated or unique. A sub-allocated file is one which is defined using space from one or more existing data spaces. For such a file, DLBL and EXTENT statements are not required. Label processing is not performed since information needed to set up the file is in the DEFINE command, and information about the data spaces to be used for the file is in the VSAM catalog.

A unique VSAM file is one which occupies data space uniquely allocated to it, not to be shared by other files. The data and the index of a key-sequenced unique file must occupy separate data spaces; each requires DLBL and EXTENT statements.

Figure 38 is an example of defining a suballocated key-sequenced file.

```
// JOB    DEFINE
// EXEC   IDCAMS,SIZE=26K
   DEFINE CLUSTER(NAME(MSTRFILE)   -
          RECORDS(100,10)   -
          VOLUME(231942)
            RECORDSIZE(40 55)   -
          FREESPACE(10 5)
            SUBALLOCATION   -
            INDEXED   -
          KEYS(8 2) UPDATEPW(WRITEFL)  -
          ATTEMPTS(0))              -
          CATALOG(VSAMCAT/SECRET)
```

Figure 38. Defining a Key-Sequenced Suballocated VSAM File

The following parameters are used in Figure 38.

- NAME (MSTRFILE) -- This parameter is required and specifies the name to be given to the file being defined.
- VOLUME (231942) -- This required parameter is used to specify the volume on which the defined object is to be placed.
- RECORDS (primary [secondary]) -- This parameter specifies the amount of space to be suballocated in terms of the number of records the space is to hold.
- RECORDSIZE (size1 size2) -- This required parameter specifies the length attributes of the logical records in the file. The size specified can be from 1 to 32,761. size1 is the average length of all logical records. size2 is the maximum length of any logical record.
- FREESPACE (percent 1 [percent 2]) -- This parameter specifies the percentage of space that is to be reserved during initial and subsequent allocations. percent 1 specifies the amount of unused space to be left in each control interval. percent 2 specifies the amount of unused control intervals be left in each control area.

  Note: This parameter is valid for key-sequenced files only.

- UNIQUE/SUBALLOCATION -- This parameter specifies whether the object is allocated a space of its own, or whether a portion of an already defined VSAM data space is suballocated to the object.

  UNIQUE
  specifies that the object being defined is allocated a space of its own. An object with the UNIQUE attribute appears in the VTOC of its volume under its own name.

  SUBALLOCATION
  specifies that a portion of an already defined VSAM data space is suballocated to the object. Objects with the SUBALLOCATION attribute do not appear in the VTOC. Only the name of the data space that contains the object appears there. If the object has the SUBALLOCATION attribute, there must be a VSAM data space defined on the volume on which the object is being defined.

- INDEXED/NONINDEXED -- This parameter specifies the type of cluster being defined.

  INDEXED
      specifies that the cluster being defined is for a key-sequenced file. This is the default.

  NONINDEXED
      specifies that the cluster being defined is for an entry-sequenced file.

- KEYS (length position) -- This parameter specifies the length and the starting position of the key field within each logical record. (Position 0 is the first byte in the logical record.) The key field with this specified length, and starting in the specified position, is in all logical records in a key-sequenced file. The sum of length and position must be equal to or less than the length of the logical record.

- UPDATEPW (password) -- This parameter specifies the update level password for the file being defined. The update level password permits input and output operations (READ, START, DELETE, WRITE, REWRITE) against the logical records of the file.

Note that this file has no read-level protection and that its master level password is WRITEFL.

ATTEMPTS (count)
    specifies the maximum number of times the operator can try to enter the password in response to a prompting message. Count can be any number from 0 through 7. The value 0 prevents any password prompting.

CATALOG (catalog name/password)
    specifies the catalog and its update level password that is to contain the entries for the cluster.

File Processing Techniques

The COBOL user has three different file processing techniques available to him; sequential, random, and a combination of sequential and random. The technique to be used is specified through the ACCESS clause of the SELECT statement.

Entry-Sequenced File Processing: An entry-sequenced file can only be processed sequentially; therefore, since the default is sequential, the ACCESS clause need not be specified.

Key-Sequenced File Processing: A key-sequenced file can be processed sequentially, randomly, or both sequentially and randomly. To process sequentially, ACCESS IS SEQUENTIAL is specified. To process randomly, ACCESS IS RANDOM is specified. To process both sequentially and randomly, ACCESS IS DYNAMIC is specified.

ACCESS IS DYNAMIC provides the greatest flexibility since all the capabilities of both sequential and random processing are supported. Processing can be switched from sequential to random and vice-versa, as many times as desired.

Current Record Pointer

The current record pointer (CRP), a conceptual pointer, is applicable only to key-sequenced files. The current record pointer indicates the next record to be accessed by a sequential request; the CRP has no meaning for random processing. The CRP is affected only by the OPEN, START and READ statements, it is not used or affected by the WRITE, REWRITE, or DELETE statements. The following are examples of how the CRP is affected by various COBOL statements.

Example 1:

Assuming a file has records with keys from 1 to 10, if the sequence of I/O operations on the file with ACCESS IS DYNAMIC and opened I-O is:

```
MOVE 7 TO RECORD-KEY
READ filename
MOVE 44 TO RECORD-KEY
WRITE record-name
READ filename NEXT RECORD
```

the READ NEXT reads record 8 if the previous READ was successful. If the previous READ was not successful, the STATUS KEY will be set to 94 (No Current Record Pointer) when the READ NEXT is attempted. This occurs independently of the successful intervening WRITE.

Generally, the last request on a file which establishes a CRP (OPEN, READ, or START) must have been successful in order for a sequential read to be successful.

Example 2:

In this example, ACCESS IS SEQUENTIAL is specified; therefore, records are retrieved in ascending key sequence starting at the position indicated by the CRP. (Assume this file has records with keys from 1 to 10.)

```
OPEN INPUT filename         (CRP is at first
                            record on the
                            file)

MOVE 10 TO RECORD-KEY

START filename              (CRP is now at
                            record 10)

READ filename               (record 10 is
                            read)

MOVE 5 TO RECORD-KEY

START filename              (CRP is now at
                            record 5)

READ filename               (record 5 is read
                            CRP is set to
                            record 6)

READ filename               (record 6 is read
                            CRP is set to
                            record 7)
```

Note that the CRP can be changed randomly through the use of the START statement. All reading is then done sequentially from that point. In this example, if the START request for record key 5 had failed with no record found (File Status=23), the three READ statements following would have failed with no current record pointer (File Status=94).

Example 3:

In this example ACCESS IS DYNAMIC is specified. Therefore, records are accessed randomly if READ is specified and sequentially if READ NEXT is specified. (Assume this file has records with keys from 1 to 44.)

```
OPEN INPUT             (CRP is set to first
                       record on file)

MOVE 5 TO RECORD-KEY

READ filename          (record 5 is read, CRP
                       is set to record 6)

READ filename          (record 6 is read, CRP
NEXT RECORD            is set to record 7)
(or indent a couple
of spaces)

Move 41 TO RECORD-KEY

READ filename          (record 7 is read, CRP
NEXT RECORD            is set to record 8)
(or indent a couple
of spaces)
```

The last READ---NEXT RECORD does not read record 41 even though the record key field contained 41. This is true because a sequential read does not use the contents of the record key to determine which record to read, it uses the position of CRP as established by a previous request. If the last READ had been a random read (no NEXT) then record 41 would have been read.

Example 4:

In this updating example, ACCESS IS DYNAMIC is specified; the REWRITE statement does not affect the CRP. (Assume this file has records with keys from 1 to 44.)

```
OPEN I-O               (CRP is at first
                       record on file)

MOVE 10 TO RECORD-KEY

READ filename          (record 10 is read,
                       CRP is set at record
                       11)

MOVE 44 TO RECORD-KEY

REWRITE record-name    (record 44 is updated,
                       CRP is set at record 11)

READ filename          (record 11 is read, CRP
NEXT RECORD            is set at record 12)

MOVE 74 TO RECORD-KEY

REWRITE                (fails, record not
                       found in this file)

READ NEXT              (record 12 is read,
                       CRP is set at record 13)
```

Note that although the last REWRITE failed, the following READ NEXT was successful.

Table 11. File Status Values and Error Handling

	No USE Declarative		USE Declarative	
First Character of FILE STATUS	AT END or INVALID KEY clause	No AT END or INVALID KEY clause	AT END or INVALID KEY clause	No AT END or INVALID KEY clause
0	Return to next sentence	Return to next sentence	Return to next sentence	Return to next sentence
1	Return to AT END address	Return to next sentence	Return to AT END address	Return to next sentence after USE declarative is executed
2	Return to INVALID KEY address	Return to next sentence	Return to INVALID KEY address	Return to next sentence after USE declarative is executed
3	Write message and return to next sentence	Write message and return to next sentence	Return to next sentence after USE declarative is executed	Return to next sentence after USE declarative is executed
9	Return to next sentence	Return to next sentence	Return to next sentence after USE declarative is executed	Return to next sentence after USE declarative is executed

ERROR HANDLING

All errors on a VSAM file, whether logic errors caused by the COBOL programmer (for example, reading an unopened file), or I-O errors on the external storage media, return control to the COBOL program. The contents of FILE STATUS indicate the status of the last request on the file. It is strongly recommended that all files have a file status associated with them, and that the COBOL programmer check the contents of FILE STATUS after each request.

Table 11 describes the actions taken for all the combinations of AT END, INVALID KEY, and error declaratives for each value of FILE STATUS.

Note: Return is always to NEXT STATEMENT unless the request that caused the error contained an AT END or INVALID KEY clause. By omitting both the AT END and INVALID KEY clauses and the USE ERROR/EXCEPTION for the file, any type of error for the file can be intercepted by checking the FILE STATUS data name following each I/O request (including OPEN and CLOSE) for the file. This will simplify the exception-condition handling in the COBOL program.

Record Formats for VSAM Files

For VSAM files, processing is independent of whether or not the records on a file are fixed-length (that is, all records in the file are the same length) or of variable-length format.

Thus for example, the considerations which are discussed in "Record Formats For Non-VSAM Files" generally do not apply.

However, the following points should be considered:

- For record handling purposes, the records are considered to be fixed-length when

  1. All the records in the file are the same size (or there is only one record description).

  2. No record contains an OCCURS clause with the DEPENDING ON option.

  Otherwise, the records are considered to be variable length.

- For variable length records, without OCCURS DEPENDING ON clauses, the following applies:

When a READ INTO statement is used, the size of the longest record for the file is moved to the input area. Coding considerations for records with the OCCURS DEPENDING ON option are discussed in "Table Handling Considerations."

Initial Loading of Records into a File

A non-loaded file is one which has been defined but has never contained any records. An unloaded file is one which has contained records but from which all records have been deleted. A loaded file is one which contains records.

Initial loading is the process of writing records into a non-loaded file.

It is strongly recommended that initial loading of records into a key-sequenced file be done sequentially. If the initial loading is done randomly, performance will be slower, not only for the initial loading process, but also for all processing done on that file later on. Random loading of records does not reserve free space in the file; therefore, the file will be dynamically reorganized when any subsequent records are inserted.

The following table illustrates which OPEN options are allowed for each file state.

OPEN OPTION \ FILE STATE	NON-LOADED	UNLOADED	LOADED
INPUT	NO	YES	YES
OUTPUT	YES	NO	NO
I-O	NO	YES	YES
EXTEND	YES	YES	YES

From this table it can be seen that opening a file with the OUTPUT option is valid only when the file is new (has never contained any records). Also, opening a file with the INPUT or I-O option is valid only when the file is not new. If such a file contains no records (is in the unloaded state) the first READ request results in an AT END condition (if ACCESS IS SEQUENTIAL) or an INVALID KEY condition (if ACCESS IS RANDOM or DYNAMIC).

File Status Initialization

The value of 'Z' in Status Key 1 is reserved for the programmer's use. This permits his determining whether a request was made against his file. For example, if he initializes Status Key 1 to the value Z before attempting to OPEN his file he can then determine if his program actually attempted the OPEN by checking the contents of Status Key 1. If it is Z, the OPEN statement was not executed; if it is a value other than Z, the statement was executed. This same technique can be used for any request against the file (CLOSE, READ, etc.) to determine if such a request was attempted in his program.

Opening a VSAM File

If any of these rules are violated, the file is not opened and the FILE STATUS key is set to the appropriate value. Refer to Table 12 for FILE STATUS key values at open time. Table 13 describes file status at action request time.

A loaded file can be opened EXTEND, INPUT, or I-O. If such a file is opened EXTEND and it is a key-sequenced file, the first record to be added must have its record key higher than the highest record key on the file when it was opened. If it is not higher, a logic error results, and the FILE STATUS key is equal to 92. For an entry-sequenced file, the records are added after the last record.

Since the USE declarative is executed only for files that are in open status, the only OPEN error which can cause the USE DECLARATIVE to be invoked is trying to open a file which is already in the open status. This is a logic error and causes file status to be set to 92. The open status of the file is not affected. However, if the file is defined as ACCESS IS DYNAMIC, the illegal OPEN statement causes the current record pointer to be undefined.

40 Null file opened Input or I-O [illegible]

Table 12. File Status Key Values at OPEN

File Status	Probable Cause
30	I-O error
91	Incorrect password. Either an incorrect password was specified or a required password was not specified. If a file is opened OUTPUT, EXTEND, or I-O, the UPDATE password is required.
92	Logic error caused by opening an opened file, or by opening a locked file.
93	Resource not available. Caused by insufficient virtual storage, or the file is not available for the type of processing requested.[1]
95	Invalid or incomplete information in the ASSGN card, or the file was not found in the catalog.[2]
96	Missing DLBL card

[1]Indicates that the file was already opened by someone else and opening it for this request would violate the share options specified for the file.

[2]FILE STATUS 95 can also be caused by the following:

- an attempt to open a key-sequenced file as if it were an entry-sequenced file or vice versa.
- an attempt to open a non-loaded file with the INPUT or I-O option.
- an attempt to open OUTPUT a file not in the non-loaded state.
- record key length or displacement specification that does not match what was specified when the file was defined.

Table 13. File Status at Action Request Time

File Status		Probable Cause
00		Successful
10		A sequential READ statement encountered EOF.
21	key-sequenced file only	A request was issued to change the record key during execution of a REWRITE statement, or a sequence error occurred for a sequentially-accessed key-sequenced file.
22	key-sequenced file only	A request was issued to add a record whose record key was a duplicate of a record already on the file.
23	key-sequenced file only	Either a READ statement was issued for a record whose record key does not match any record on the file, or a REWRITE or DELETE statement was issued for a record not on the file.
24	key-sequenced file only	A request was issued to write a record beyond the externally-defined boundaries of the file.
30		An I-O error occurred.
34		A request was issued to write a record beyond the externally-defined boundaries of an entry-sequenced file.
92		A logic error occurred. (See Note below.)
93		Resource not available. Insufficient virtual storage or volume, extent unavailable, or data already in exclusive control.
94		No current record pointer for a sequential READ statement.
99		Abnormal termination (subroutine error).

Note: File Status = 92 can be caused by the following:

- Any request issued against an unopened file.
- Any request issued which is not allowed for the OPEN option; for example, issuing a READ statement for a file opened OUTPUT, or a REWRITE statement for a file opened INPUT.
- Any attempt to write or rewrite a record longer than the maximum record size specified when the file was defined.
- Any action taken on a file after EOF has been encountered (entry-sequenced or key-sequenced file). If EOF is encountered on a key-sequenced file, a START or a READ statement can be issued to reset the CRP and continue processing. For example, a key-sequenced file with ACCESS IS SEQUENTIAL specified:

```
OPEN
READ     successful
READ     EOF encountered
READ     logic error
START    reset CRP
READ     successful
```

  or, a key-sequenced file with ACCESS IS DYNAMIC specified:

```
OPEN
READ NEXT     successful
READ NEXT     EOF encountered
READ NEXT     logic error
READ          reset CRP (random READ)
READ NEXT     successful
```

- An attempt to rewrite when ACCESS IS SEQUENTIAL has been specified if the preceding action was not a successful READ operation.
- An attempt to delete when ACCESS IS SEQUENTIAL was specified if the preceding action was not a successful READ operation (key-sequenced file only).
- An attempt to read with improper length specified.

WRITING RECORDS INTO A VSAM FILE

The COBOL WRITE statement is used to add a record to a file. (Existing records in the file are not replaced with this statement.) The record to be written must not be larger than the maximum record size specified when the file was defined.

Entry-Sequenced File Considerations for the WRITE Statement

Entry-sequenced file records are written sequentially. If the file is not opened OUTPUT or EXTEND, FILE STATUS is set to 92 and the record is not written.

Key-Sequenced File Considerations for the WRITE Statement

When ACCESS IS SEQUENTIAL is specified, the file must be opened OUTPUT or EXTEND. If not, the WRITE statement is not executed and FILE STATUS is set to 92.

The records must be written in ascending key sequence. If the file is opened EXTEND, the record keys of the records to be added must be higher than the highest record key on the file when it was opened. The following example shows the action and resultant FILE STATUS when a file containing records whose keys are 2, 4, 6, 8, and 10 is opened EXTEND. (Refer to Table 13 explanations of FILE STATUS values at action request time.)

ACTION	FILE STATUS
WRITE (record key = 8)	92
WRITE (record key = 9)	92
WRITE (record key = 12)	00
WRITE (record key = 11)	21
WRITE (record key = 6)	21

Note that the first two WRITE requests result in a logic error (FILE STATUS=92) because their key values are not higher than the highest key on the file when it was opened. Once a successful WRITE has taken place all subsequent WRITE requests are handled as though the file were opened OUTPUT. This is why the WRITE of record key 6 causes a sequence error, not a logic error.

If many records are to be added to a file, it is strongly recommended that sequential access be used. Performance is improved both for the process of adding the records and for later retrieval of them.

When ACCESS IS RANDOM or ACCESS IS DYNAMIC is specified, the file must be opened I-O or OUTPUT. If not, the WRITE statement is not executed and FILE STATUS is set to 92. The records can be written in any order.

REWRITING RECORDS ON A VSAM FILE

The COBOL REWRITE statement is used to replace existing records on the file.

Entry-Sequenced File Considerations for the REWRITE Statement

For successful REWRITE statement execution, the file must be opened I-O. The record to be rewritten must first be read by the COBOL program, then updated by the REWRITE statement. (The length of the record being rewritten cannot be changed.) If there was no preceding READ statement, or if the preceding READ statement was not successful (EOF was reached), the REWRITE statement is not executed and FILE STATUS is set to 92.

Key-Sequenced File Considerations for the REWRITE Statement

For successful REWRITE statement execution, the file must be opened I-O. The length of the record can be changed, but the value of the record key cannot be changed.

When ACCESS IS SEQUENTIAL is specified, the record to be rewritten must first be read by the COBOL program, then updated by the REWRITE statement. The REWRITE statement is not successful if the preceding statement for the file was not a successful READ of this record. This causes file status to be set to 92.

When ACCESS IS RANDOM or ACCESS IS DYNAMIC is specified, the record does not need to be read by the COBOL program. The record is updated by moving its key to the record key field and doing the REWRITE.

READING RECORDS ON A VSAM FILE

The COBOL READ statement is used to access records on a file. If the file is not opened INPUT or I-O, the READ statement is not executed and FILE STATUS is set to 92.

Entry-Sequenced File Considerations for the READ Statement

Records are read sequentially, in the order in which they were written.

Key-Sequenced File Considerations for the READ Statement

When ACCESS IS SEQUENTIAL is specified, records are read sequentially, beginning at the position of the current record pointer. If the current record pointer is undefined when the READ is executed, FILE STATUS is set to 94. The following example shows successful and unsuccessful READ and START executions. (Assume this file has records with keys 1 through 8 and 20.)

OPEN I-O filename	CRP at first record on file
READ file name	(first record on file is read)
MOVE 10 TO RECORD-KEY	
START file name	(fails-no record found)
READ file name	(fails-no CRP)
MOVE 20 TO RECORD-KEY	
START file name	(successful)
READ file name	(record 20 is read)

When ACCESS IS RANDOM is specified, records are read in the order specified by the program. To read records whose record key is 10, move 10 to the RECORD KEY field in the record area and issue a READ statement.

When ACCESS IS DYNAMIC is specified, records can be read randomly or sequentially. The READ NEXT statement is used for sequential accessing, and the READ statement is used for random accessing.

READ NEXT Statement

Records are read sequentially beginning at the position of the current record pointer. If the current record pointer is not defined when the READ NEXT statement is issued, FILE STATUS is set to 94 as a result of the READ. The current record pointer is considered undefined if the preceding START or READ statement was not successful.

For details on the effect of COBOL statements on the position on the current record pointer, refer to the section entitled "Current Record Pointer."

READ Statement

The READ statement reads records randomly using the value placed in the record key field.

USING THE START VERB

The START statement is only valid for key-sequenced files but not when ACCESS IS RANDOM is specified or when the file is opened OUTPUT or EXTEND.

In some of the preceding examples, the START verb was used to position the CRP. Then the READ (for ACCESS IS SEQUENTIAL) and READ NEXT (for sequential processing when ACCESS IS DYNAMIC) retrieves the record pointed to by the CRP as established by the START.

Example:

```
     05  RECORD-KEY.
         10  GEN11.
             15  GEN12  PIC 99.
             15  GEN13  PIC 99.
         10  GEN14  PIC9.
```

In this example, GEN12, GEN11, or RECORD-KEY could be used as the data-name in the "KEY IS relational data-name" option of the START statement. The lengths would be 2, 4, and 5 respectively. GEN13 and GEN14 could not be used as they are not in the leftmost part of RECORD-KEY.

Assume that the value of RECORD-KEY is 01472:

- START filename KEY = GEN11 would position the CRP to the first record on the file whose key has 0147 as the first 4 characters.

- START file-name KEY > GEN12 would position the CRP to the first record in the file whose key has the first two characters greater than 01.

DELETE Statement

The DELETE is valid only for a key-sequenced File. The same considerations discussed under "Key-Sequenced File Considerations for the REWRITE Statement" apply to the DELETE statement.

COBOL Language Usage With VSAM

The COBOL language statements which are directly related to VSAM processing are in the section "DOS/VS COBOL Considerations" in the publication IBM DOS Full American National Standard COBOL. The following paragraphs are intended only to highlight and summarize the basic language statements used in writing a VSAM-file-processing COBOL program.

A COBOL programmer can use VSAM in three basic ways: to create a file, to retrieve a file, and to update a file. However, prior to processing a VSAM file, it is an absolute necessity that the previously discussed Access Method Services functions be performed. Most significant to the COBOL programmer is whether the file is defined as an entry-sequenced file or as a key-sequenced file.

Creating a VSAM File

The minimum COBOL language statements required to create a VSAM file are summarized in Table 14.

Table 14. COBOL Statements for Creating a VSAM File

	Entry-Sequenced File	Key-Sequenced File
Environment Division	SELECT ASSIGN	SELECT ASSIGN ORGANIZATION IS INDEXED RECORD KEY
Data Division	FD entry LABEL RECORDS	FD entry LABEL RECORDS
Procedure Division	OPEN OUTPUT or OPEN EXTEND WRITE CLOSE	OPEN OUTPUT or OPEN EXTEND WRITE CLOSE

The following discussion illustrates the steps which must be taken to create an entry-sequenced file. Assume the VSAM catalog and VSAM data space have been created as previously illustrated. The next thing a user must do is define the entry in the catalog for the VSAM file.

```
// JOB   DEFINE FILE
// EXEC  IDCAMS,SIZE=100K
     DEFINE  CLUSTER(NAME(TRANFILE)       -
       VOLUME(321942)  RECORDS(50 5)      -
       RECORDSIZE(80 80)  READPW(R0104)   -
       UPDATEPW(W0104)  ATTEMPTS(0)       -
       NONINDEXED SUBALLOCATION)          -
     CATALOG(VSAMCAT/SECRET)
/*
```

The meaning of the parameters is:

NAME (TRANFILE)	This is the data set name.
VOLUME (321942)	This is the volume on which the space for the data set resides.
RECORDS (50 5)	Primary allocation is for 50 records, secondary allocation is for 5 records.
RECORDSIZE (80 80)	The average and maximum record size is 80 characters.
READPW (R0104)	The password R0104 must be supplied to open the file with the INPUT option.
UPDATEPW (W0104)	The password W0104 must be supplied to open the file with the OUTPUT, EXTEND or I-O option.
ATTEMPTS(0)	The operator is not to be prompted for the password when the file is opened.
NONINDEXED	The file is an entry-sequenced file.
SUBALLOCATION	Space for this file is to be suballocated from existing VSAM data space on the volume.
CATALOG (VSAMCAT/ SECRET)	The name of the catalog into which this file is cataloged is VSAMCAT and its update password is SECRET.

Note: When the user gains update access to the file (by supplying the update level of the password) he has also gained read access. In general, when a user gains access to a file at a given level of protection, he has gained access to that file for all lower levels. This means that the above file could be opened INPUT by supplying the update level of the password. However, it could not be opened OUTPUT, EXTEND or I-O by supplying the read level password.

The COBOL program to access such a file would include the following statements.

```
FILE-CONTROL.
     SELECT VSAMSEQ
          ASSIGN TO SYS010-AS-TESTFL
          ORGANIZATION IS SEQUENTIAL
          ACCESS IS SEQUENTIAL
          PASSWORD IS VSAMPW
          FILE STATUS IS STATKEY.
          '
          '
DATA   DIVISION.
FILE   SECTION.
FD     VSAMSEQ
     LABEL RECORDS ARE OMITTED.
01     VSAMREC.
       05      FIELD1     PICTURE X(8).
       05      FIELD2     PICTURE X(72).
          '
          '
          '
WORKING-STORAGE SECTION.
77     STATKEY            PICTURE 99.
77     VSAMPW             PICTURE X(5).
          '
          '
          '
PROCEDURE DIVISION.
BUILD-PASSWORD.
       PERFORM  PASSWORD-BUILDER.
       PERFORM  PASSWORD-SCRAMBLER.
          '
          '
          '
       OPEN  OUTPUT  VSAMSEQ.
       IF STATKEY NOT = 0
               GO TO ERROR-HANDLER.
BUILD-A-RECORD.
          '
          '
          '
       WRITE  VSAMREC.
       IF STATKEY NOT = 0
               GO TO ERROR-HANDLER.
          '
          '
          '
       GO TO BUILD-A-RECORD.
          '
          '
          '
```

In this sample program the routines PASSWORD-BUILDER and PASSWORD-SCRAMBLER construct the update level password so that the file can be opened OUTPUT. These routines can be written in such a way that they are difficult to follow, thus improving security.

Note that the FILE-STATUS is checked after each request on the file. This ensures that unexpected conditions will be detected.

The JCL needed to execute the program is

```
//      JOB
//      ASSGN    SYS010 X'130'
//      DLBL     TESTFL,'TRANFILE',,VSAM
//      EXTENT   SYS010,321942
//      EXEC     program-name,SIZE=nnnk
```

Example 2:

This example shows the creation of a COBOL key-sequenced VSAM file. This program performs the same function as example 1 except that now a key-sequenced file is being created. The records in the file "INREC" are in ascending key order.

```
IDENTIFICATION DIVISION.
       .
       .
       .
ENVIRONMENT DIVISION.
       .
       .
       .
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INREC
      ASSIGN TO SYS005-UR-2540R-CARDIN.
    SELECT OUTREC
      ASSIGN TO SYS010-OUTMAST
      ORGANIZATION IS INDEXED
      RECORD KEY IS ARG-1
      FILE STATUS IS CHK.
          .
          .
          .
DATA DIVISION.
FILE SECTION.
FD  INREC LABEL RECORDS ARE OMITTED
    DATA RECORD IS INMASTER
01  INMASTER PIC X(80).
FD  OUTREC LABEL RECORDS ARE STANDARD
    DATA RECORD IS OUTMASTER.
01  OUTMASTER.
    05  FILLER PIC X.
    05  ARG-1 PIC XXX.
    05  REM PIC X(76).
WORKING-STORAGE SECTION.
77  CHK PIC XX.
```

```
PROCEDURE DIVISION.
PARA1.
    OPEN INPUT INREC OUTPUT OUTREC.
    IF CHK IS NOT = "00" GO TO CHKRTN.
PARA2.
    READ INREC INTO OUTMASTER
         AT END GO TO PARA4.
PARA3.
    WRITE OUTMASTER.
    IF CHK IS NOT = "00" GO TO CHKRTN.
    GO TO PARA2.
PARA4.
    CLOSE INREC OUTREC.
    IF CHK IS NOT = "00" GO TO CHKRTN.
FINIT.
    CLOSE INREC OUTREC.
    STOP RUN.
CHKRTN.
    DISPLAY "ERROR.  STATUS KEY VALUE
     IS" CHK
    GO TO FINIT.
```

Note that in this example any Status Key return other than 00 causes transfer of control to paragraph CHKRTN. This routine can determine the exact cause of the error by checking the Status Key. Once the cause is determined, instructions can be issued according to the user's desired response to each type of error.

Retrieving a VSAM File

The minimum COBOL language statements required to retrieve a VSAM file are summarized in Table 15.

Table 15. COBOL Statements for Retrieving a VSAM File

	Entry-Sequenced File	Key-Sequenced File
Environment Division	SELECT ASSIGN	SELECT ASSIGN ORGANIZATION IS INDEXED RECORD KEY
Data Division	FD entry LABEL RECORDS	FD entry LABEL RECORDS
Procedure Division	OPEN INPUT READ ... AT END CLOSE	OPEN INPUT READ CLOSE

The following examples show the retrieval of records from VSAM files.

Example 3:

This example shows the retrieval of records from the entry-sequenced file created in example 1. The records are then printed.

```
IDENTIFICATION DIVISION.
       .
       .
       .
ENVIRONMENT DIVISION
       .
       .
       .
INPUT-OUTPUT SECTION.
FILE-CONTROL
    SELECT INREC
    ASSIGN TO SYS010-AS-INMAST
    FILE STATUS IS CHK.
    SELECT PREC
    ASSIGN TO SYS005-UR-1403-S-PRNTR
          .
          .
          .
DATA DIVISION.
FILE SECTION.
FD  INREC LABEL RECORDS ARE STANDARD
    DATA RECORD IS INMASTER.
01  INMASTER PIC X(80).
FD  PREC LABEL RECORDS ARE OMITTED
    DATA RECORD IS POUT.
01  POUT PIC X(80).
WORKING-STORAGE SECTION.
77  CHK PIC XX.
PROCEDURE DIVISION.
PARA1.
    OPEN INPUT INREC OUTPUT PREC.
    IF CHK IS NOT = "00" GO TO CHKRTN.
PARA2.
    READ INREC INTO POUT AT END GO TO
      PARA4.
    IF CHK IS NOT = "00" GO TO CHKRTN
PARA3.
    WRITE POUT.
    GO TO PARA2.
PARA4.
    CLOSE OUTREC PREC.
    IF CHK IS NOT = "00" GO TO CHKRTN.
FINIT.
    STOP RUN.
CHKRTN.
    DISPLAY 'ERROR.  STATUS KEY VALUE
      IS' CHK.
    GO TO FINIT.
```

Note that in this example any Status Key return other than 00 causes transfer of control to paragraph CHKRTN. This routine can determine the exact cause of the error by checking the Status Key. Once the cause is determined, instructions can be issued according to the user's desired response to each type of error.

Example 4:

This example shows the retrieval of records from the key-sequenced file created in example 2. Note that in the Procedure Division there is a switch from sequential processing to random processing; this is permitted since ACCESS IS DYNAMIC is specified in the ENVIRONMENT Division.

```
IDENTIFICATION DIVISION.
       .
       .
       .
ENVIRONMENT DIVISION.
       .
       .
       .
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INREC
      ASSIGN TO SYS010-INMAST
      ORGANIZATION IS INDEXED
      ACCESS IS DYNAMIC
    RECORD KEY IS ARG-1
      FILE STATUS IS CHK.
    SELECT PREC
      ASSIGN TO SYS005-UR-1403-S-PRINTR
          .
          .
DATA DIVISION.
FILE SECTION.
FD  INREC LABEL RECORDS ARE STANDARD
    DATA RECORD IS INMASTER.
01  INMASTER.
    05  FILLER PIC X.
    05  ARG-1  PIC XXX.
    05  ARG-2  PIC XX.
    05  ARG-3  PIC XX.
    05  FILLER PIC X(72).
FD  PREC LABEL RECORDS ARE OMITTED
    DATA RECORD IS POUT.
01  POUT PIC X(80).
WORKING-STORAGE SECTION.
77  CHK PIC XX.
PROCEDURE DIVISION.
PARA1.
    OPEN INPUT INREC OUTPUT PREC.
    IF CHK IS NOT = "00" GO TO CHKRTN.
PARA2.
    MOVE "003" TO ARG-1.
    START INREC.
PARA3.
    READ INREC NEXT RECORD AT END GO TO
      PARA4.
    IF CHK IS NOT = "00" GO TO CHKRTN.
    IF ARG-2 IS = "02" GO TO PARA4.
    IF ARG-3 IS NOT = "73" GO TO PARA3.
    WRITE POUT FROM INMASTER.
    GO TO PARA3.
PARA4.
    MOVE "101" TO ARG-1.
    READ INREC INVALID KEY GO TO CHKRTN.
    WRITE POUT FROM INMASTER.
    MOVE "103" TO ARG-1.
    READ INREC INVALID KEY GO TO CHKRTN.
    WRITE POUT FROM INMASTER.
```

```
PARA5.
    CLOSE INREC PREC.
      IF CHK IS NOT = "00" GO TO CHKRTN.
FINIT.
      STOP RUN.
CHKRTN.
    DISPLAY 'ERROR.  STATUS KEY VALUE
      IS' CHK.
    GO TO FINIT.
```

Note that in this example any Status Key return other than 00 causes transfer of control to paragraph CHKRTN. This routine can determine the exact cause of the error by checking the Status Key. Once the cause is determined, instructions can be issued according to the user's desired response to each type of error.

Job Control Language for a VSAM File

JCL is simplified for VSAM since all VSAM files must be cataloged through Access Method Services.

The JCL to execute the program in example 1 is

```
// JOB
// ASSGN    SYS010,X'233'
// DLBL     OUTMAST,'PAYFILE',,VSAM
// EXTENT   SYS010,VSAMVOL
// EXEC     EXAMPLE,SIZE=50K
```

The volume on which the VSAM file was defined is mounted at address 233, the volume ID is VSAMVOL, and the file was given the name PAYFILE when it was defined. The SIZE parameter is required on the EXEC card for VSAM programs.

Converting Non-VSAM Files to VSAM Files

ISAM files can be converted to VSAM files so that they may be processed by a COBOL program using VSAM. The conversion is done through Access Method Services.

Essentially, the conversion process consists of defining a VSAM file as the target for the file being converted. Then through the appropriate JCL and the REPRO command, the conversion is accomplished.

For a complete description of the conversion process, see DOS/VS Utilities VSAM Access Method Services, and DOS/VS Data Management Guide.

Using ISAM Programs to Process VSAM Files

Once the file is converted the programmer can process the new VSAM file with his old ISAM program by converting his ISAM JCL to VSAM JCL. For more details on this procedure see DOS/VS Data Management Guide.

The following topics are discussed within this chapter:

COBOL VSAM Control Blocks

DTF Tables

Error Recovery for Non-VSAM Files

Volume and File Label Handling

COBOL VSAM CONTROL BLOCKS

The compiler generates a File Information Block (FIB) from information in the Environment Division (SELECT, RERUN, and SAME statements) and the Data Division (FD and associated records). The File Control Block (FCB) is generated dynamically at execution time by the VSAM library subroutines. The user may wish to refer to fields in these blocks for debugging. The format of the VSAM control block (Access Method Control Block -- ACB) is not given here, as the knowledge of its contents is not needed by the COBOL user.

CONTROL BLOCKS FOR VSAM

The following two control blocks are required to process input/output requests for VSAM files.

VSAM FILE INFORMATION BLOCK (FIB)

The file information block, a portion of the completed object module, is used at execution time by the ILBDINT0, ILBDVOC0, and ILBDVIO0 COBOL library subroutines for processing input/output verbs used with VSAM files. The FIB is built by phase 21.

Fixed Portion:

Displacement			No. of	
Hex	Decimal	Field	Bytes	Description
0	0	IFIBID	1	FIB identification code: X'I'
1	1	IFIBLVL	1	FIB level number
2	2	INAMED	7	External name
9	9	INAMEDB	1	External name
A	10		1	Reserved
B	11	IORG	1	ORGANIZATION

Code:

Bits	Equate Name	Bit Settings	Meaning
0-7	IORGVPS	1000 1000	VSAM ADDRESSED SEQUENTIAL
	IORGVIX	0100 1000	VSAM INDEXED

Hex	Decimal	Field	Bytes	Description
C	12	IACCESS	1	ACCESS MODE

Code:

Bits	Equate Name	Bit Settings	Meaning
0-7	IACCSEQ	1000 0000	SEQUENTIAL
	IACCRAN	0100 0000	RANDOM
	IACCDYN	0010 0000	DYNAMIC

Hex	Decimal	Field	Bytes	Description
D	13		1	Reserved
E	14	ISW1	1	Miscellaneous switches

Code:

Bits	Equate Name	Bit Settings	Meaning
0-7	ISOPTNL	1000 0000	OPTIONAL specified
	ISSAMREC	0010 0000	SAME RECORD AREA specified

Hex	Decimal	Field	Bytes	Description
F	15		1	Reserved
10	16		6	Reserved
16	22	IRECLEN	2	Number of bytes in longest 01-entry
18	24	IRECDBL	2	Displacement in TGT of record's first base locator cell
1A	26	IRECNBL	1	Number of base locators for RECORD AREA
1B	27		1	Reserved
1C	28	ISTATDBL	2	Displacement in TGT of base locator for STATUS data-name
1E	30	ISTATDDN	2	Displacement from base locator of STATUS data-name
20	32	ISTATLDN	2	Length of STATUS data-name
22	34		1	Reserved
23	35	IKEYNO	1	Number of entries in key list

24	36	IKEYFNTL	2	Length of each entry in key list
26	38	IPSWISW	1	Miscellaneous switches
27	39	IPSWNO	1	Number of entries in password list
28	40	IPSWENTL	2	Length of each entry in password list
2A	42		14	Reserved
38	56	IMISCAD	4	Address in variable length portion of FIB for miscellaneous clauses
3C	60	ILABELAD	4	Reserved
40	64	IKEYLSTA	4	Address of first key list entry
44	68	IPSWLSTA	4	Address of first password list entry
48	72		16	Reserved

Variable Length Portion:

Supplementary information for miscellaneous clauses (one for each clause):

Displacement			No. of	
Hex	Decimal	Field	Bytes	Description
0	0	IMSW1	2	Switch bytes

Code:

Bits	Equate Name	Bit Settings	Meaning
0-7	IMRREOV	1000 0000	RERUN at end of volume
8-15			Reserved

2	2	IRERUNI	4	RERUN integer (field contains zeros if RERUN not specified)
6	6		2	Slack bytes
8	8	IRERUNN	8	External-name of RERUN clause

Key List Entry: (one per user-defined key--RECORD/ALTERNATE/RELATIVE)

Displacement			No. of	
Hex	Decimal	Field	Bytes	Description
0	0	KEYSW	1	Miscellaneous switches

Code:

Bits	Equate Name	Bit Settings	Meaning
0-7	IKEYCOMP	1000 0000	Key is USAGE COMP (binary)

1	1	IKEYLDN	1	Length of key data-name
2	2	IKEYDBL	2	Displacement of key data-name's locator in TGT
4	4	IKEYDDN	2	Data-name displacement from locator

Password List Entry: (one per password)

0	0	IPSWDIXN	1	Associated index number 0 = none 1 = primary
1	1	IPSWDLDN	1	Length of password data-name
2	2	IPSWDDBL	2	Displacement of password data-name's locator in TGT
4	4	IPSWDDDN	2	Data-name displacement from locator

VSAM FILE CONTROL BLOCK (FCB)

The VSAM File Control Block is created by the ILBDINTO COBOL library subroutine. It is used by the ILBDVIOO and ILBDVOCO subroutines to interface with the VSAM system control subroutines

Displacement Hex	Displacement Decimal	Field	No. of Bytes	Description
0	0	FCBID	1	FCB identification code: 'F'
1	1	FCBLVL	1	FCB level number
2	2	FOPENOPT	4	Save area for OPEN options
6	6	FCLOSOPT	4	Save area for CLOSE options
A	10		2	Reserved
C	12	FCOBRTN	4	Address of COBOL transmitter routine
10	16	FUSERR	4	Address of USE...ERROR declarative
14	20	FUSELIST	4	Address of USE declarative Exit List
18	24		6	Reserved
1E	30	FRECKEY	1	Number of RECORD KEY
1F	31	FADVANC	1	Reserved
20	32	FENDINV	4	Return address from INVALID KEY, AT END, or end-of-page
24	36		12	Reserved for compilation-dependent fields
30	48	FOPENOPS	4	Options for VSAM OPEN verb

Code:

Bits	Equate Name	Bit Settings	Meaning
0-7	FOPIN	1000 0000	INPUT
	FOPOUT	0100 0000	OUTPUT
	FOPIO	0010 0000	I-O
	FOPEXT	0001 0000	EXTEND
8-15	Reserved		
16-23	FOPUERR	1000 0000	USE...ERROR declarative address in FUSERR cell
24-31	Reserved		

Displacement Hex	Displacement Decimal	Field	No. of Bytes	Description
34	52	FCLOSOPS	4	VSAM CLOSE options

Code:

Bits	Equate Name	Bit Settings	Meaning
0-7	FCLLOCK	0001 0000	LOCK
8-31	Reserved		

Displacement Hex	Displacement Decimal	Field	No. of Bytes	Description
38	56	FSW1	4	Miscellaneous switches

Code:

Bits	Equate Name	Bit Settings	Meaning
0-7	FSOPEN	1000 0000	File is open
	FSLOCKED	0100 0000	File is closed with lock
	FSOPTNL	0010 0000	Optional file not present
	FSOKACT	0001 0000	Successful action has occurred since open
	FSEOF	0000 1000	Sequential read has encountered end-of-file
	FSVCORE	0000 0100	Main storage to process this open has been acquired
8-31			Reserved

3C	60	FTRSTMT	4	Transmission statement switches

Code:

Bits	Equate Name	Bit Settings	Meanings
0-7	FTREAD	0000 0100	READ statement
	FTWRITE	0000 1000	WRITE statement
	FTREWRT	0000 1100	REWRITE statement
	FTSTART	0001 0000	START
	FTDELET	0001 0100	DELETE statement
8-15	FTINVKEY	1000 0000	INVALID KEY
	FTATEND	0100 0000	AT END
	FTNEXT	0000 0010	NEXT
	FTKEY	0000 0001	KEY
16-23	FTSRCHGT	1000 0000	GREATER THAN
	FTSRCHEQ	0100 0000	EQUAL TO
	FTSRCHGE	0010 0000	NOT LESS THAN
24-31	Reserved		

40	64	FSYSCBAL	4	Address of system control blocks address list
44	68	FSYSCBLL	4	Address of system control blocks lengths list
48	72	FSYSCBNO	2	Number of system control blocks (DTF, DCB, ACB)
4A	74	FKEYLEN	2	Length of KEY data-name
4C	76	FRECCNT	4	Record count for checkpoint subroutine, if RERUN specified
50	80	FFIBAD	4	Address of File Information Block (FIB)
54	84	FWORKAD	4	Address of system-dependent work area
58	88	FRECA	4	Address of current record area
5C	92	FSAMRECA	4	Address of SAME RECORD AREA
60	96	FSTATKEY	2	STATUS KEY work area
62	98	FLASTREQ	1	Last I/O statement

Code:

Bits	Equate Name	Bit Settings	Meanings
0-7	FLASTRD	0000 0100	READ statement
	FLASTWRT	0000 1000	WRITE statement
	FLASTRWT	0000 1100	REWRITE statement
	FLASTSTR	0001 0000	START statement
	FLASTDLT	0001 0100	DELETE statement
	FLASTOPN	0001 1000	OPEN statement
	FLASTCLO	0001 1100	CLOSE statement

63	99		13	Reserved

DTF TABLES

Whenever COBOL imperative-statements (READ, WRITE, REWRITE, etc.) are used in a program to control the input and/or output of records in a file, that file must be defined by a DTF. A DTF is created by the compiler for each file opened in a COBOL program from information specified in the Environment Division, FD entry, and input/output statements in the source program. The DTF for each file is part of the object module that is generated by the compiler. It describes the characteristics of the logical file, indicates the type of processing to be used for the file, and specifies the storage areas and routines used for the file.

The DTF's generated for the permissible combinations of device type and COBOL file processing technique are as follows:

DTFCD	Card reader, punch -- organization and access sequential
DTFPR	Printer -- organization and access sequential
DTFMT	Tape -- organization and access sequential
DTFSD	Mass storage device -- organization and access sequential
DTFDA	Mass storage device -- organization direct, access sequential or random
DTFIS	Mass storage device -- organization indexed, access sequential or random
DTFDU	3540 diskette -- organization and access sequential

Because of their limited interest for the COBOL programmer, the contents and location of the fields of each of the DTF types are not discussed in this publication. However, there are certain fields which immediately precede the storage area allocated for the DTF which are pertinent. These fields are provided on the listing in hexadecimal if an abnormal termination occurs and the SYMDMP option is in effect. The SYMDMP option is described in detail in the chapter "Symbolic Debugging Features." Fields preceding the DTF are described below.

For magnetic tape files (DTFMT) or sequentially organized files on mass storage devices (DTFSD), a 26-byte Pre-DTF is reserved in front of the DTF. The fields of the Pre-DTF are shown in Table 16. If any option is not specified, the field will contain binary zeros.

When actual track addressing is used for files with direct organization and random access (DTFDA), a variable-length Pre-DTF is reserved. The fields of the Pre-DTF are shown in Table 23. If any option is not specified, the field will contain binary zeros.

When relative track addressing is used for files with direct organization and random access (DTFDA), a variable-length Pre-DTF is reserved. The fields of the Pre-DTF are shown in Table 18. If any option is not specified, the field will contain binary zeros.

Table 16. Fields Preceding DTFMT and DTFSD

2 bytes	Length of nonstandard label, if present
1 byte	Number of reels (as specified in the ASSIGN clause) when file is opened[1]
1 byte	Number of reels remaining (i.e., file not completely read)[1]
2 bytes	Maximum record length if records are variable, blocked and APPLY WRITE-ONLY is not specified.
4 bytes	Address of label declarative with BEGINNING REEL UNIT option
4 bytes	Address of label declarative with ENDING REEL UNIT option
4 bytes	Address of label declarative with ENDING FILE option
4 bytes	Address of label declarative with BEGINNING FILE option
1 byte	Switch -- FF if closed WITH LOCK; otherwise, the switch is used as shown in Table 23
3 bytes	Address of USE AFTER STANDARD ERROR declarative
	DTFMT/DTFSD
[1]For INPUT files with nonstandard labels only.	

Table 17. Fields Preceding DTFDA -- ACCESS IS RANDOM -- Actual Track Addressing

9-263 bytes	ACTUAL KEY[1]
8 bytes	SEEK Address[2]
2 bytes	Error bytes[3]
4 bytes	Address of file extent information
4 bytes	Address of label declarative with ENDING FILE option
4 bytes	Address of label declarative with BEGINNING FILE option
1 byte	Switch -- FF if closed WITH LOCK; otherwise the switch is used as shown in Table 23
3 bytes	Address of USE AFTER STANDARD ERROR declarative
	DTFDA
[1]ACTUAL KEY specified in last executed WRITE statement [2]In the form MBBCCHHR [3]This area is reserved by the Supervisor and assigned the name ERRBYTE. For a complete discussion, refer to the publication DOS/VS Supervisor and I/O Macros, Order No. GC24-5037.	

Table 18. Fields Preceding DTFDA -- ACCESS IS RANDOM -- Relative Track Addressing

5-258 bytes	ACTUAL KEY[1]
4 bytes	SEEK address[2]
3 bytes	Last extent used[3]
1 byte	Not used
2 bytes	Error bytes[4]
1 byte	Index to last extent used in the Disk Extent Table
3 bytes	Address of Disk Extent Table in the DTF
4 bytes	Address of label declarative with ENDING FILE option
4 bytes	Address of label declarative with BEGINNING FILE option
1 byte	Switch -- FF if closed WITH LOCK; otherwise the switch is used as shown in Table 23
3 bytes	Address of USE AFTER STANDARD ERROR declarative
DTFDA	

[1]ACTUAL KEY specified in the last executed WRITE statement
[2]In the form TTTR
[3]In the form TTT
[4]This area is reserved by the DOS/VS Supervisor and assigned the name ERRBYTE. For a complete discussion, refer to the publication DOS/VS Supervisor and I/O Macros.

When actual track addressing is used for files with direct organization and sequential access (DTFDA), a 31-byte Pre-DTF is reserved. The fields of the Pre-DTF are shown in Table 19. If any option is not specified, the field will contain binary zeros.

When relative track addressing is used for files with direct organization and sequential access (DTFDA), a 31-byte Pre-DTF is reserved. The fields of the Pre-DTF are shown in Table 20. If any option is not specified, the field will contain binary zeros.

For files whose organization is indexed, eight bytes are reserved preceding the DTF, as shown in Table 21. The fields preceding the DTFDU for the 3540 are shown in Table 22.

Table 19. Fields Preceding DTFDA -- ACCESS IS SEQUENTIAL -- Actual Track Addressing

8 bytes	SEEK address[1]
5 bytes	IDLOC[2]
2 bytes	Error bytes[3]
4 bytes	Address of file extent information
4 bytes	Address of label declarative with ENDING FILE option
4 bytes	Address of label declarative with BEGINNING FILE option
1 byte	Switch -- FF if closed WITH LOCK; otherwise the switch is used as shown in Table 23
3 bytes	Address of USE AFTER STANDARD ERROR declarative
	DTFDA

[1]In the form MBBCCHHR
[2]Address (returned by the system) of next record in the form CCHHR
[3]This area is reserved by the DOS/VS Supervisor and assigned the name ERRBYTE. For a complete discussion, refer to the publication DOS/VS Supervisor and I/O Macros.

Table 20. Fields Preceding DTFDA -- ACCESS IS SEQUENTIAL -- Relative Track Addressing

4 bytes	SEEK address[1]
3 bytes	Last extent used[2]
1 byte	Not used
4 bytes	IDLOC[3]
1 byte	Not used
2 bytes	Error bytes[4]
1 byte	Index to the last extent used in the Disk Extent Table
3 bytes	Address of Disk Extent Table in the DTF
4 bytes	Address of label declarative with ENDING FILE option
4 bytes	Address of label declarative with BEGINNING FILE option
1 byte	Switch -- FF if closed with LOCK; otherwise the switch is used as shown in Table 23
3 bytes	Address of USE AFTER STANDARD ERROR declarative
DTFDA	

[1]In the form TTTR
[2]In the form TTT
[3]Address (returned by the system) of the next record in the form TTTR
[4]This area is reserved by the DOS/VS Supervisor and assigned the name ERRBYTE. For a complete discussion, refer to the publication _DOS/VS Supervisor and I/O Macros_.

Table 21. Fields Preceding DTFIS

2 bytes	Unused
2 bytes	Displacement of record key within record
1 byte	Switch -- FF if closed WITH LOCK; otherwise the switch is used as shown in Table 23
3 bytes	Address of USE AFTER STANDARD ERROR declarative
DTFIS	

Table 22. Fields Preceding DTFDU

4 bytes	Unused
1 byte	DTF switch -- FF if closed with LOCK
3 bytes	Address of USE AFTER STANDARD ERROR declarative
DTFDU	

Some files can be opened several different ways in one COBOL program.

For DTFCD and DTFPR, only one DTF will be generated for each file.

For DTFMT, a maximum of three DTF's may be needed -- one each for OPEN INPUT, OPEN INPUT REVERSED, and OPEN OUTPUT.

For DTFSD, a maximum of three DTF's may be needed -- one each for OPEN INPUT, OPEN OUTPUT, and OPEN I-O statements.

For DTFIS and DTFDA, only one DTF is needed.

Pre-DTF Switch

When used, this switch provides communication between the executing program and its input/output subroutines at execution time. The entire byte may be set to X'FF' to indicate that the file was closed WITH LOCK and cannot be reopened. Otherwise the switch is used as shown in Table 23.

Table 23. Meaning of Pre-DTF Switch

Bit	Meaning, if ON
0	Turned ON the first time a DTFSD output file is opened. The entire DTF is saved for subsequent OPEN OUTPUT statements.
1	Turned ON when DTFDA or DTFSD files are opened I-O.
2	This bit is ON to indicate beginning of volume user label processing. The bit is set OFF when a file is opened to indicate to the user label processing subroutine (ILBDUSL0) that beginning-of-file user labels are to be processed. That subroutine sets the bit ON after beginning-of-file processing to indicate that all subsequent calls for this subroutine are for beginning-of-volume user label processing.
3	For output files with variable-length blocked records, this bit is turned OFF when a file is opened and ON for all WRITE's after the first.
4	Turned ON for spanned record processing on a DTFDA file.
5-6	Not used.
7	New ILBDIML0-Indicator for Rel. 2.5 and higher (see note below).

Note: This bit is set by Rel. 2.5 ILBDIML0 and tested by transient $$BFCMUL. If this bit is not on, exit is taken immediately, because PUB pointer in DTF-8 is incorrect.

ERROR RECOVERY FOR NON-VSAM FILES

COBOL allows the programmer to handle input/output errors through 1) the INVALID KEY clause for certain source language statements, and 2) the USE AFTER STANDARD ERROR declarative sentence.

Input/output errors caused by the program can be recovered from directly by the procedure specified in the INVALID KEY clause. That is, when the system determines that an invalid key condition exists, control is returned to the programmer at the imperative-statement specified in the INVALID KEY clause. An invalid key condition can occur on files with direct or indexed organization and on sequentially organized disk files. The errors that cause an invalid key condition are shown in Table 24.

Table 24. Errors Causing an Invalid Key Condition

Organization	ACCESS	OPEN	I-O Verb	Condition
Sequential	[SEQUENTIAL]	OUTPUT	WRITE	End of extents reached.
Direct	[SEQUENTIAL]	OUTPUT	WRITE	Track address outside file extents.
Direct	RANDOM	INPUT	READ	No record found.
		OUTPUT	WRITE	Track address outside file extents.
		I-O	READ REWRITE	Track address outside file extents.
Indexed	[SEQUENTIAL]	INPUT I-O	START	No record found.
		OUTPUT	WRITE	Duplicate record; sequence check.
	RANDOM	INPUT	READ	No record found.
		I-O	REWRITE	
		I-O	WRITE	Duplicate record.

Other input/output errors cause the job to be cancelled unless the programmer has specified a USE AFTER STANDARD ERROR declarative. Control is transferred to this declarative section if the system determines that a "standard" error has occurred during input/output processing. In this declarative section, the programmer may interrogate the COBOL error bytes if he has specified the GIVING option of the USE AFTER STANDARD ERROR declarative sentence. The meaning of these bytes for a specified combination of device type and file processing technique is shown in Table 25.

Table 25. Meaning of Error Bytes for GIVING Option of Error Declarative (Part 1 of 2)

Device	Organization	ACCESS	OPEN	I/O Verb	Condition	Byte	Result
Unit record	Sequential	[SEQUENTIAL]			Input/output error	1	File must be closed and job must be terminated.
Tape	Sequential	[SEQUENTIAL]	INPUT	READ	Wrong length record	2	Skip block if return is made to non-declarative portion.
					Parity error	1	Skip block if return is made to non-declarative portion.
			OUTPUT	WRITE	All exceptional conditions are handled by the system.		
DASD	Sequential	[SEQUENTIAL]	INPUT I-O	READ	Wrong length record	2	Skip block if return is made to non-declarative portion.
					Parity error	1	Skip block if return is made to non-declarative portion.
			OUTPUT I-O	WRITE	Parity error	1	Bad block written.
					Wrong length record	2	Bad block written.
DASD	Direct	[SEQUENTIAL]	INPUT	READ	Wrong length record	2	Return to statement after READ.
					Data check in count area	1	Return to statement after READ.
					Data check for key and/or data	4	Return to statement after READ.
DASD	Direct	RANDOM	INPUT I-O	READ	Same as ACCESS SEQUENTIAL (above).		
			OUTPUT	WRITE	Wrong length record	2	Return to next statement; bad block written.
					Data check in count area	1	Return to next statement; bad block written.
					Data check for key and/or data	4	Return to next statement; bad block written.
					No room found	3	Return to next statement.

<u>Note</u>: If no USE AFTER STANDARD ERROR routine is specified and one of the above conditions occurs, the programmer is notified of the condition and the job is cancelled.

Table 25. Meaning of Error Bytes for GIVING Option of Error Declarative (Part 2 of 2)

Device	Organization	ACCESS	OPEN	I/O Verb	Condition	Byte	Result
DASD	Direct	RANDOM	I/O	REWRITE	Wrong length record	2	Return to next statement; bad block written.
					Data check in count area	1	Return to next statement; bad block written.
					Data check in key and/or data	4	Return to next statement; bad block written.
DASD	Indexed	[SEQUENTIAL]	INPUT I-O	READ REWRITE	DASD error	1	Return to next statement; bad block read or written.
					Wrong length record	2	
				START	DASD error	1	Continued processing of file permitted.
			OUTPUT	WRITE	DASD error	1	Return to next statement; bad block written.
					Wrong length record	2	
					Prime data area full	3	File must be closed.
					Cylinder index full	4	File must be closed.
					Master index full	5	File must be closed.
DASD	Indexed	RANDOM	INPUT I-O	READ REWRITE	DASD error	1	Return to next statement; bad block read or written.
					Wrong length record	2	
			I-O	WRITE	DASD error	1	Return to next statement; bad block written.
					Wrong length record	2	
					Overflow area full	6	Files must be closed.
3540	Sequential	Sequential	INPUT	READ	Data check	1	Return to next statement.
			OUTPUT	WRITE	Equipment check	2	Bad block read or written up until bad physical record.

Note: If no USE AFTER STANDARD ERROR routine is specified and one of the above conditions occurs, the programmer is notified of the condition and the job is cancelled.

If the programmer includes a USE AFTER STANDARD ERROR routine without specifying the GIVING option, he must call an assembler language routine within the declarative if he wishes to interrogate the error bits -- set either in the DTF (DTFMT, DTFSD, or DTFIS) or in the fields preceding the DTF (DTFDA).

Interrogation of these error bits should be made to the locations shown in Tables 26, 27, 28, 29, and 30.

Note: The byte and bit displacement in Tables 26, 27, 28, 29, and 30 is relative to zero.

Table 26. Location and Meaning of Error Bits for DTFMT

OPEN	Verb	Condition	Byte*	Bit
INPUT	READ	Wrong length record	3	1
		Parity error	2	6
OUTPUT	WRITE	Wrong length record	3	1
		Parity error	2	6
*Within the DTF.				

Table 27. Location and Meaning of Error Bits for DTFSD

OPEN	Verb	Condition	Byte*	Bit
INPUT, I-O	READ	Wrong length record	3	1
		Parity error	2	6
OUTPUT, I-O	WRITE	Parity error	2	6
*Within the DTF.				

Table 28. Location and Meaning of Error Bits for DTFDA

ACCESS	OPEN	Verb	Condition	Byte*	Bit
[SEQUENTIAL]	INPUT	READ	Wrong length record	0	1
			Data check in count area	1	0
			Data check in key or data	1	3
			No record found	1	2 or 4
RANDOM	INPUT, I-O	READ	Same as sequential		
	OUTPUT	WRITE	Wrong length record	0	1
			No room found	0	4
			Data check in count area	1	0
			Data check in key or data	1	3
	I-O	REWRITE	Wrong length record	0	1
			Data check in count area	1	0
			Data check in key or data	1	3
			No record found	1	2 or 4
*Within error bytes preceding DTF. See the section "DTF Tables" for the location of these bytes.					

Table 29. Location and Meaning of Error Bits for DTFIS

ACCESS	OPEN	Verb	Condition	Byte*	Bit
[SEQUENTIAL]	INPUT, I-O	READ	DASD error	30	0
			Wrong length record	30	1
	OUTPUT	WRITE	DASD error	30	0
			Wrong length record	30	1
			Prime data area full	30	2
			Cylinder index full	30	3
			Master index full	30	4
RANDOM	INPUT, I-O	READ REWRITE	DASD error	30	0
			Wrong length record	30	1
	I-O	WRITE	DASD error	30	0
			Wrong length record	30	1
			Overflow area full	30	6
*Within the DTF.					

Table 30. Location and Meaning of Error Bits for DTFDU

ACCESS	OPEN	Verb	Condition	Byte*	Bit
Sequential	Input	READ	Data check	3	3
	Output	WRITE	Equipment check	2	2

The following should be considered when processing tape input files:

1. Two types of errors are returned to the programmer: wrong length record and parity check. The COBOL error bytes, if requested, are set to reflect the error condition and control is transferred to the USE AFTER STANDARD ERROR declarative sentence. The error block is made available at data-name-2 of the GIVING option, if specified.

   If a parity error is detected when a block of records is read, the tape is backspaced and reread 100 times before control is returned to the programmer. If the error persists, the block is considered an error block and is added to the block count found in the DTF table.

2. Normal return (to the non-declarative portion) from a USE AFTER STANDARD ERROR declarative section is through the invoked IOCS subroutine. Thus, the next sequential block is brought into storage permitting continued processing of the file. (The error block is bypassed.) A return through the use of a GO TO statement does not bring the next block into storage; therefore, it is impossible to continue processing the file.

The processing of a sequential disk file opened as input is the same as the previous discussion of tape files, except that the disk block is reread ten times before being considered an error block.

COBOL cannot handle nested errors on sequential files. If errors occur within an error declarative, results are unpredictable.

VOLUME AND FILE LABEL HANDLING

TAPE LABELS

Among the several types of tape labels allowed under the Disk Operating System Virtual Storage are: volume labels, standard file labels, user standard labels, and nonstandard labels. Unlabeled files are also permitted. The description of each type of label follows.

Volume Labels

A volume label is used whenever standard file labels are used. Logical IOCS requires a volume label with VOL1 as its first four characters on every standard or user standard labeled file. VOL2-VOL8 are also allowed, but must be written by the programmer and are only used by OS.

Standard File Labels

A standard file label is an 80-character label created when an output file is opened or closed, in part by IOCS using the TLBL control statement. The first three characters are HDR (header), EOV (end-of-volume), or EOF (end-of-file). The fourth character is a 1, indicating the first of a possible eight labels. The remainder of the label is formatted into fields describing the file. Labels 2 through 8 in this field are bypassed on input, and are not created on output under the Disk Operating System Virtual Storage.

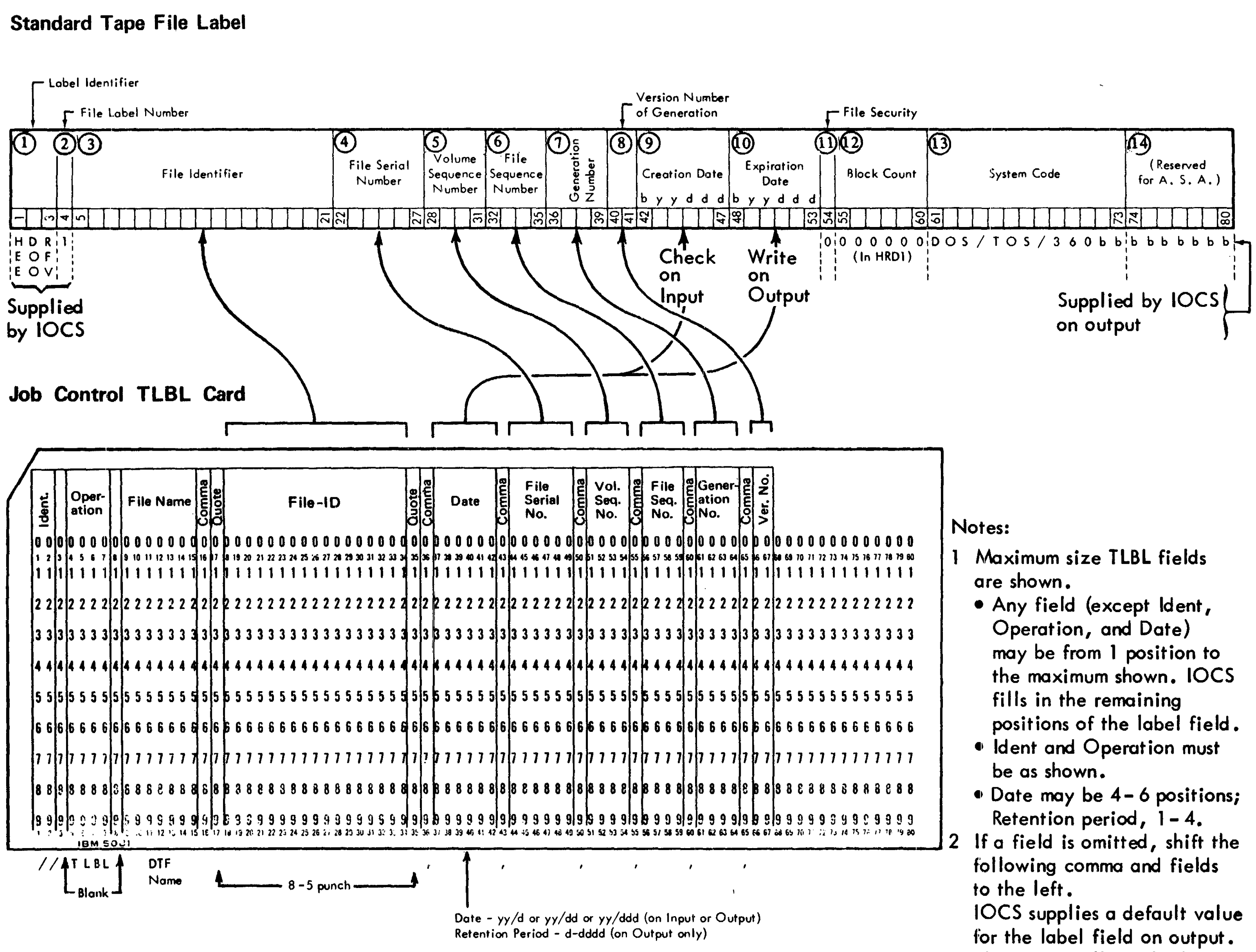
The contents of the fields of a standard file label are described in "Appendix B: Standard Tape File Labels." The relationship between the TLBL statement and a standard file label is shown in Figures 39 and 40.

User Standard Labels

A user standard label is an 80-character label having UHL (user header label) or UTL (user trailer label) in the first three positions. The fourth position contains a number 1 through 8 which represents the relative position of the user label within a group of user labels. The contents of the remaining 76 positions are entirely up to the programmer. User labels, if present, follow HDR, EOV, or EOF standard labels. On multivolume files, they may also appear at beginning-of-volume. User header labels are resequenced starting with one (UHL1) at the beginning of a new volume. Figure 41 shows the positioning of user labels on a file.

Nonstandard Labels

A nonstandard label may be any length. The contents of a nonstandard label is entirely programmer-dependent. It is the COBOL programmer's responsibility either to process or bypass nonstandard labels on input and to create them on output. Nonstandard label processing is not permitted on ASCII files. Figure 42 shows the positioning of nonstandard labels on a file.



Notes:

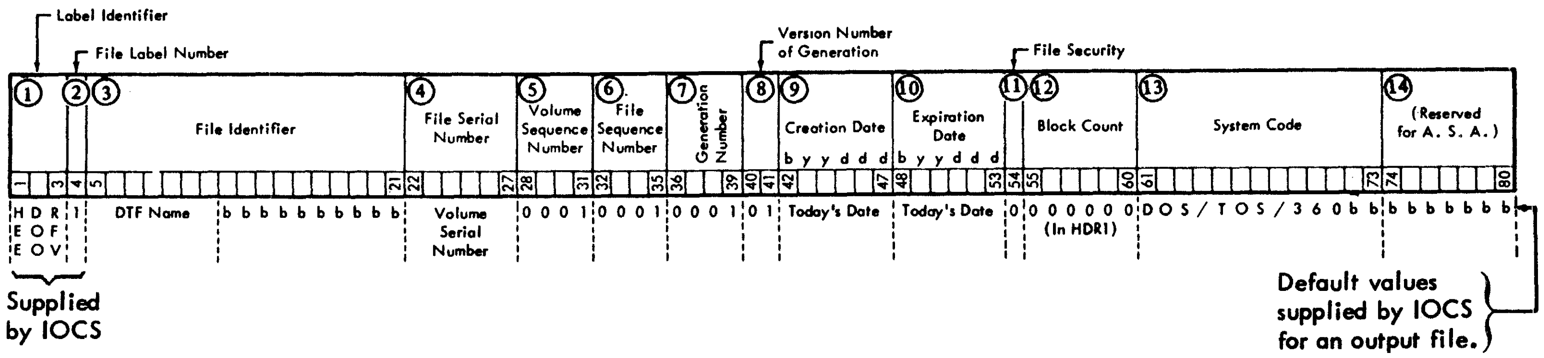
1 Maximum size TLBL fields are shown.
- Any field (except Ident, Operation, and Date) may be from 1 position to the maximum shown. IOCS fills in the remaining positions of the label field.
- Ident and Operation must be as shown.
- Date may be 4-6 positions; Retention period, 1-4.

2 If a field is omitted, shift the following comma and fields to the left. IOCS supplies a default value for the label field on output.

3 No comma follows the last field used.

Figure 39. Standard Tape File Label and TLBL Card (Showing Maximum Specifications)

Standard Tape File Label



On input, no values are supplied and no checking is performed.

Job Control TLBL Card

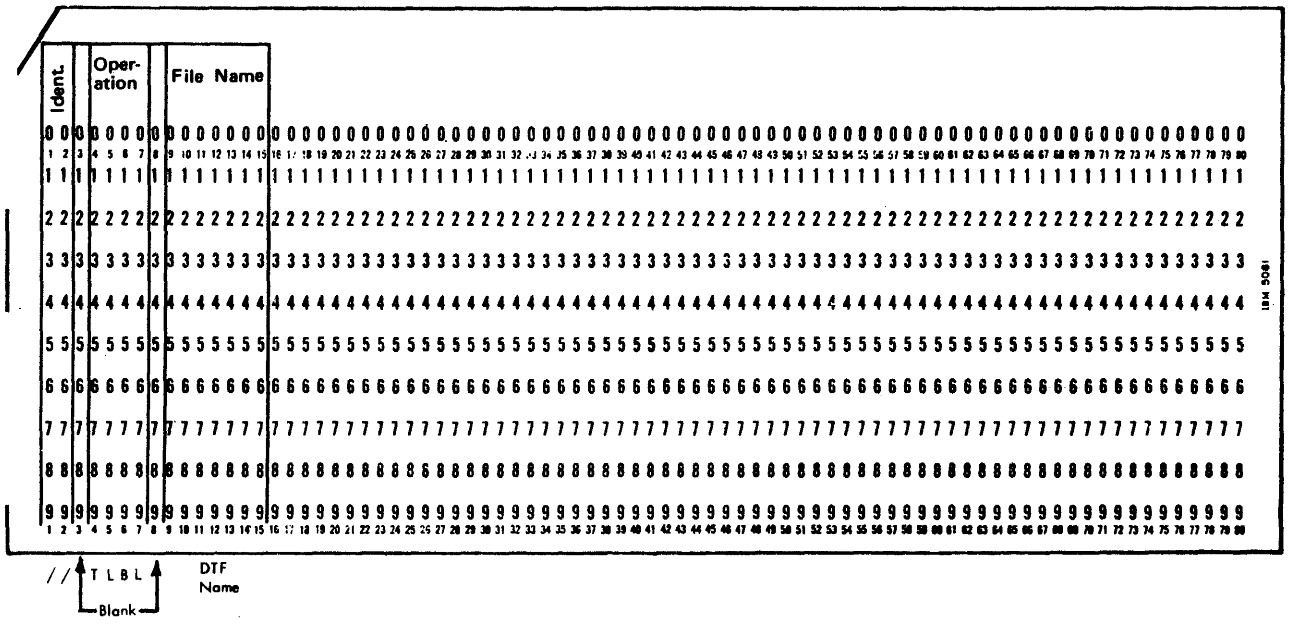


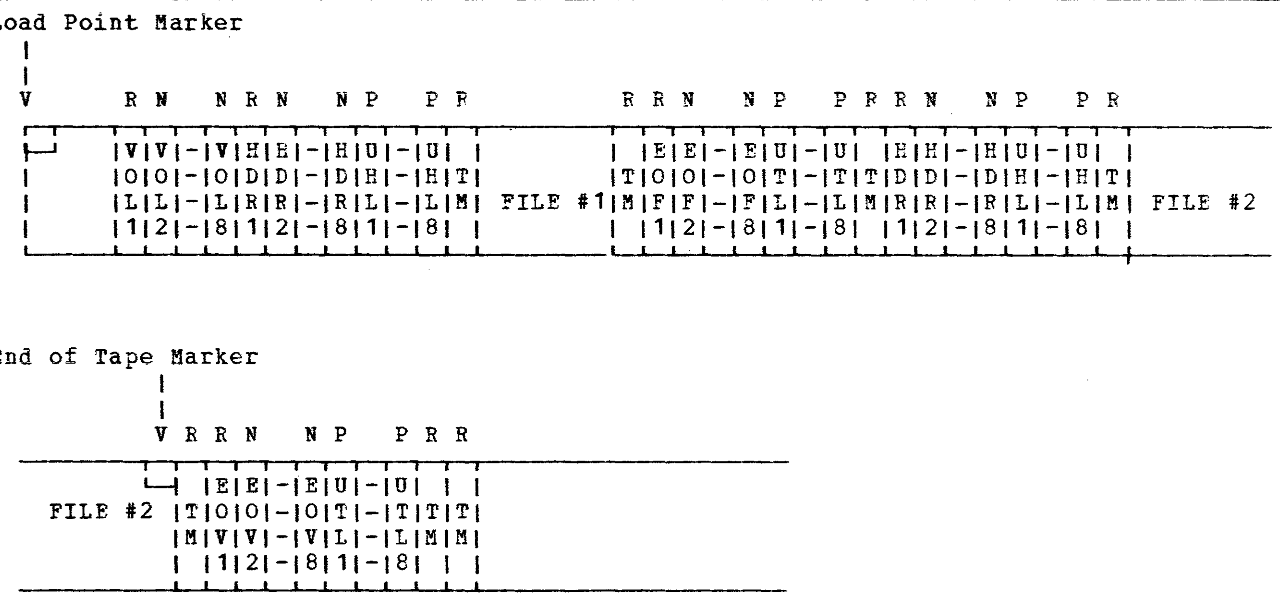
Figure 40. Standard Tape File Label and TLBL Card (Showing Minimum Requirements)

LABEL PROCESSING CONSIDERATIONS

Label considerations for VSAM are discribed in the chapter "Virtual Storage Access Method (VSAM)".

The labels which may appear on tape are shown in Figures 40 and 41. The compiler allows the programmer to work with files containing all the previously mentioned labels as well as with unlabeled files.

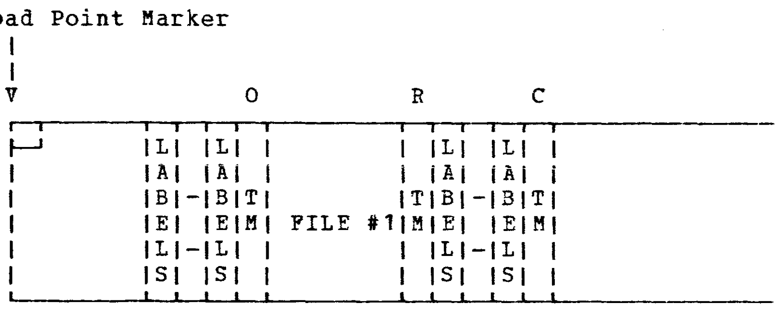
If user standard labels are to be created or checked in the COBOL program, the USE AFTER BEGINNING/ENDING LABELS declarative sentence and the LABEL RECORDS clause with the data-name option must be specified.



Load Point Marker
V
R N N R N N P P R
VOL1 VOL2 - VOL8 HDR1 HDR2 - HDR8 UHL1 - UHL8 TM
FILE #1
R R N N P P R R N N P P R
TM EOF1 EOF2 - EOF8 UTL1 - UTL8 TM HDR1 HDR2 - HDR8 UHL1 - UHL8 TM
FILE #2
End of Tape Marker
V R R N N P P R R
FILE #2
TM EOV1 EOV2 - EOV8 UTL1 - UTL8 TM TM

Notes: R = Required, processed by IOCS.
N = Permitted, but not written or checked, by IOCS and not available to programmer.
P = Processed by IOCS and available to user.

Figure 41. Standard, User Standard, and Volume Labels



Notes: R = Required, processed by IOCS.
O = Optional.
C = Written by COBOL compiler.

Figure 42. Nonstandard Labels

Header labels are written or read when the file is opened or when a volume switch occurs. Trailer labels are written when the physical end of the reel is reached, or when a CLOSE REEL or CLOSE file-name is issued. Trailer labels are read on each reel except the last when a tapemark is reached. For the last reel (i.e., EOF labels), trailer labels are not read until the file is closed.

For multivolume input files with nonstandard labels, the programmer must specify the integer-1 option of the source language ASSIGN clause, where integer-1 is the number of reels in the file. This number can be overridden at execution time by storing a nonzero integer in the special register NSTD-REELS before opening the file. The number of reels is then available to the programmer while the file is opened both in the special register NSTD-REELS and in the field reserved for this purpose which precedes the DTF table for DTFMT (see "DTF Tables" in this chapter). In addition, the number of reels remaining after each volume switch can also be found in the field reserved for this purpose which precedes the DTF table for DTFMT.

When processing a multivolume file with nonstandard labels (i.e., when the data-name option of the LABEL RECORDS clause is specified), if the programmer wishes to stop reading or writing before the physical end of a reel is reached, he must set a switch in the appropriate declarative section. In the Procedure Division, he can either CLOSE REEL or CLOSE FILE depending on the switch setting. Volume switching is done by LIOCS when CLOSE REEL is executed.

Note: An unlabeled multivolume tape file should not be CLOSE WITH LOCK between two reels.

Sample Programs

Figure 43 illustrates the manner in which unlabeled input files on a multifile volume are processed by a COBOL program. The input volume contains four files, only three of which are being used by the program. This unused file, which resides between the first and third file on the volume, must be bypassed during file processing. The program creates a single multivolume file with standard labels.

(1) All input files residing on the same volume are assigned to the same symbolic unit.

(2) The second file on the input reel is not used in this program and is bypassed through use of the POSITION option of the MULTIPLE FILE TAPE clause.

(3) The first and second input files are closed by the execution of the CLOSE statement with the NO REWIND option, leaving the tape positioned in mid-reel for the next OPEN.

(4) All volumes with the exception of the last volume of the multivolume output file are closed by a close statement with the REEL option. Volume switching is performed as noted in Step (8).

(5) The second and third input files processed by the program are opened by an OPEN statement with the NO REWIND option.

(6) At job completion, a standard CLOSE is issued to reposition the tapes of the closed files at their physical beginnings.

(7) An LBLTYP control statement is included because a tape file requiring label information is to be processed.

(8) Alternate assignments have been made for SYS011. Because these alternate assignments are in the sequence in which the ASSGN statements are submitted, the first volume of the output file will be on tape drive 282, the second on 283, and the third on 181. When the first CLOSE OUT-PUT REEL statement is executed, a standard EOV label is written on the volume assigned to drive 282 and the reel is rewound and positioned at its physical beginning. The next WRITE RECO statement executed will then be written on the volume mounted on drive 283.

(9) Although the file OUT-PUT consists of multiple volumes, only one TLBL control statement need be submitted.

Figure 44 is a sample program that illustrates the manner in which the multivolume file created in Figure 43 is read as an input file. The sample program also creates a multifile volume with standard labels.

(1) All output files residing on the same volume are assigned to the same symbolic unit.

The name field of the system-name in the ASSIGN clause is specified. This is the external-name by which the file is known to the system. When specified, it is the name that appears in the filename field of the DLBL or TLBL job control statements.

(2) For the multivolume input file IN-PUT, the AT END option of the READ statement applies only to the last volume containing the EOF label. For prior volumes containing EOV labels, automatic volume switching will take place as indicated in the ASSGN control statements pertaining to the file IN-PUT.

(3) The first and second file written on the volume are closed using the NO REWIND option of the CLOSE statement. This option leaves the tape positioned in mid-reel following the EOF label of the file just closed.

(4) At job's completion, a standard CLOSE is issued to reposition the tapes of the closed files at their physical beginning.

(5) A LBLTYP control statement is included because tape files requiring label information are being processed.

(6) There are three TLBL control statements for the volume assigned to SYS013, one for each file referenced on the volume. The filename field of the TLBL control statements for these files contains the names used in the ASSIGN clauses of the COBOL source program, not the programmer logical unit name.

(7) Alternate assignments have been made for SYS012 to handle the multiple volumes of the file IN-PUT.

Figure 45 illustrates the creation of an unlabeled multivolume file. The number of output volumes is determined dynamically during program execution. The program's input consists of the labeled multifile volume created in Figure 44.

(1) All input files residing on the same volume are assigned to the same symbolic unit. The name field of the system-name of the ASSIGN clause is specified. These names will appear on the TLBL control statements that refer to these files.

The MULTIPLE FILE TAPE clause is not required for the multifile volume because each file is being processed in the sequence in which it appears on the reel. A rewind will not be executed for any file on the reel except for that processed last.

(2) The CLOSE statement for files IN-PUT-1 and IN-PUT-2, and the OPEN statement for files IN-PUT-2 and IN-PUT-3, use the NO REWIND option. This leaves the tape positioned in mid-reel for the multifile volume's next OPEN statement.

(3) When it has been determined from the input data that a new output reel is required for the multivolume output file, a CLOSE OUT-PUT REEL statement is executed, processing is halted, and a message is issued to the operator which requests a new volume to be mounted.

(4) At job's completion, a standard CLOSE is issued to reposition the tapes of the closed file at their physical beginning.

(5) An LBLTYP control statement is included because tape files requiring label information are being processed.

(6) There are three TLBL control statements for the volume assigned to SYS014, one for each file referenced on the volume. The filename field of the TLBL control statements for these files contains the names used in the ASSIGN clauses of the source program and not the programmer logical unit names.

(7) Only one tape drive is assigned to the multivolume file OUT-PUT. Therefore, each time a volume is closed, processing must be halted and the operator informed to mount a new tape. This is illustrated in Step (3).

```
// JOB SAMPLE
*  UNLABELED MULTIFILE VOLUME TO MULTIVOLUME FILE WITH STANDARD LABELS
// OPTION LOG,DUMP,LINK,LIST,LISTX,XREF,SYM,ERRS,NODECK
// EXEC FCOBOL


000010 IDENTIFICATION DIVISION.
000020 PROGRAM-ID. SAMPLE-1.
000030 ENVIRONMENT DIVISION.
000040 CONFIGURATION SECTION.
000050 SOURCE-COMPUTER. IBM-370.
000060 OBJECT-COMPUTER. IBM-370.
000070 INPUT-OUTPUT SECTION.
000080 FILE-CONTROL.
000090     SELECT INPUT1 ASSIGN TO SYS010-UT-3410-S-FILE1.)
000100     SELECT INPUT2 ASSIGN TO SYS010-UT-3410-S-FILE2.} ①
000110     SELECT INPUT3 ASSIGN TO SYS010-UT-3410-S-FILE3.)
000120     SELECT OUT-PUT ASSIGN TO SYS011-UT-3410-S.
000130 I-O-CONTROL.
000140     MULTIPLE FILE TAPE CONTAINS INPUT1 POSITION 1 )
000150                                 INPUT2 POSITION 3 } ②
000160                                 INPUT3 POSITION 4.)
000170 DATA DIVISION.
000180 FILE SECTION.
000190 FD  INPUT1
000200     RECORD CONTAINS 80 CHARACTERS
000210     LABEL RECORD IS OMITTED.
000220 01  REC1 PIC X(80).
000230 FD  INPUT2
000240     RECORD CONTAINS 80 CHARACTERS
000250     LABEL RECORD IS OMITTED.
000260 01  REC2 PIC X(80).
000270 FD  INPUT3
000280     RECORD CONTAINS 80 CHARACTERS
000290     LABEL RECORD IS OMITTED.
000300 01  REC3 PIC X(80).
000310 FD  OUT-PUT
000320     RECORD CONTAINS 80 CHARACTERS
000330     BLOCK CONTAINS 3 RECORDS
000340     LABEL RECORD IS STANDARD.
000350 01  RECO PIC X(80).
000360 PROCEDURE DIVISION.
000370     OPEN INPUT INPUT1 OUTPUT OUT-PUT.
000380 READ1.
000390     READ INPUT1 INTO RECO AT END GO TO CLOSE1.
000400 A.  WRITE RECO.
000410 B.  GO TO READ1.
000420 CLOSE1.
000430     CLOSE INPUT1 WITH NO REWIND.③
000440 C.  CLOSE OUT-PUT REEL.④
000450 D.  OPEN INPUT INPUT2 WITH NO REWIND.⑤
000460 READ2.
000470     READ INPUT2 INTO RECO AT END GO TO CLOSE2.
000480     PERFORM A.
000490     GO TO READ2.
000500 CLOSE2.
000510     CLOSE INPUT2 WITH NO REWIND. ③
000520     PERFORM C.
000530     OPEN INPUT INPUT3 WITH NO REWIND. ⑤
```

Figure 43. Processing an Unlabeled Multifile Volume (Part 1 of 2)

```
000540 READ3.
000550     READ INPUT3 INTO RECO AT END GO TO CLOSE3.
000560     PERFORM A.
000570     GO TO READ3.
000580 CLOSE3.
000590     CLOSE INPUT3 OUT-PUT. ⑥
000600     STOP RUN.


// LBLTYP TAPE ⑦
// EXEC LNKEDT


// ASSGN SYS010,X'281'
// ASSGN SYS011,X'282'
// ASSGN SYS011,X'283',ALT} ⑧
// ASSGN SYS011,X'181',ALT}
// TLBL SYS011,'MULTI-VOL FILE',99/214 ⑨
// EXEC
```

Figure 43. Processing an Unlabeled Multifile Volume (Part 2 of 2)

```
// JOB SAMPLE
*  LABELED MULTIVOLUME FILE TO LABELED MULTIFILE VOLUME
// OPTION LOG,DUMP,LINK,LIST,LISTX,XREF,SYM,ERRS,NODECK
// EXEC FCOBOL


000010 IDENTIFICATION DIVISION.
000020 PROGRAM-ID. SAMPLE-2.
000030 ENVIRONMENT DIVISION.
000040 CONFIGURATION SECTION.
000050 SOURCE-COMPUTER. IBM-370.
000060 OBJECT-COMPUTER. IBM-370.
000070 INPUT-OUTPUT SECTION.
000080 FILE-CONTROL.
000090     SELECT IN-PUT ASSIGN TO SYS012-UT-3410-S.
000100     SELECT OUT-PUT1 ASSIGN TO SYS013-UT-3410-S-FILE1.}
000110     SELECT OUT-PUT2 ASSIGN TO SYS013-UT-3410-S-FILE2.} ①
000120     SELECT OUT-PUT3 ASSIGN TO SYS013-UT-3410-S-FILE3.}
000130 DATA DIVISION.
000140 FILE SECTION.
000150 FD  IN-PUT
000160     RECORD CONTAINS 80 CHARACTERS
000170     BLOCK CONTAINS 3 RECORDS
000180     LABEL RECORD IS STANDARD.
000190 01  IN-REC.
000200     05  FILLER PIC X(4).
000210     05  CODA PIC X.
000220     05  FILLER PIC X(6).
000230     05  CODB PIC X.
000240         88  SW-FIL1 VALUE "9".
000250         88  SW-FIL2 VALUE "8".
000260     05  FILLER PIC X(68).
000270 FD  OUT-PUT1
000280     RECORD CONTAINS 80 CHARACTERS
000290     BLOCK CONTAINS 3 RECORDS
000300     LABEL RECORD IS STANDARD.
000310 01  OUT-REC1 PIC X(80).
000320 FD  OUT-PUT2
000330     RECORD CONTAINS 80 CHARACTERS
000340     BLOCK CONTAINS 3 RECORDS
000350     LABEL RECORD IS STANDARD.
000360 01  OUT-REC2 PIC X(80).
000370 FD  OUT-PUT3
000380     RECORD CONTAINS 80 CHARACTERS
000390     BLOCK CONTAINS 3 RECORDS
000400     LABEL RECORD IS STANDARD.
000410 01  OUT-REC3 PIC X(80).
000420 WORKING-STORAGE SECTION.
000430 77  TAPE-NUMBER PIC 9 VALUE 0.
000440 PROCEDURE DIVISION.
000450     OPEN INPUT IN-PUT OUTPUT OUT-PUT1.
```

Figure 44. Reading a Multivolume File with Standard Labels; Creating a Multifile Volume with Standard Labels (Part 1 of 2)

```
000460 READ-IN.
000470     READ IN-PUT AT END GO TO END-OF-JOB. (2)
000480 A.  MOVE IN-REC TO OUT-REC1.
000490     WRITE OUT-REC1.
000500     IF SW-FIL1 NEXT SENTENCE ELSE GO TO READ-IN.
000510     CLOSE OUT-PUT1 WITH NO REWIND. (3)
000520     OPEN OUTPUT OUT-PUT2.
000530     ADD 1 TO TAPE-NUMBER.
000540 B.  PERFORM READ-IN.
000550     MOVE IN-REC TO OUT-REC2.
000560     WRITE OUT-REC2.
000570     IF SW-FIL2 NEXT SENTENCE ELSE GO TO B.
000580     CLOSE OUT-PUT2 WITH NO REWIND. (3)
000590     OPEN OUTPUT OUT-PUT3.
000600     ADD 1 TO TAPE-NUMBER.
000610 C.  PERFORM READ-IN.
000620     MOVE IN-REC TO OUT-REC3.
000630     WRITE OUT-REC3.
000640     GO TO C.
000650 END-OF-JOB.
000660     CLOSE IN-PUT.
000670     IF TAPE-NUMBER = 0 CLOSE OUT-PUT1 GO TO D.             }
000680     IF TAPE-NUMBER = 1 CLOSE OUT-PUT2 ELSE CLOSE OUT-PUT3. } (4)
000690 D.  STOP RUN.


// LBLTYP TAPE (5)
// EXEC LNKEDT


// ASSGN SYS018,X'283'
// TLBL FILE1,'MULTI-FILE1 VOL' }
// TLBL FILE2,'MULTI-FILE2 VOL' } (6)
// TLBL FILE3,'MULTI-FILE3 VOL' }
// ASSGN SYS012,X'281'
// ASSGN SYS012,X'282',ALT }
// ASSGN SYS012,X'181',ALT } (7)
// TLBL SYS012,'MULTI-VOL FILE'
// EXEC
```

Figure 44. Reading a Multivolume File with Standard Labels; Creating a Multifile Volume with Standard Labels (Part 2 of 2)

```
// JOB SAMPLE
*  LABELED MULTIFILE VOLUME TO UNLABELED MULTIVOLUME FILE
// OPTION LOG,DUMP,LINK,LIST,LISTX,XREF,SYM,ERRS,NODECK
// EXEC FCOBOL


000010 IDENTIFICATION DIVISION.
000020 PROGRAM-ID. SAMPLE-3.
000030 ENVIRONMENT DIVISION.
000040 CONFIGURATION SECTION.
000050 SOURCE-COMPUTER. IBM-370.
000060 OBJECT-COMPUTER. IBM-370.
000070 INPUT-OUTPUT SECTION.
000080 FILE-CONTROL.
000090     SELECT IN-PUT-1 ASSIGN TO SYS014-UT-3410-S-FILE1.)
000100     SELECT IN-PUT-2 ASSIGN TO SYS014-UT-3410-S-FILE2.} (1)
000110     SELECT IN-PUT-3 ASSIGN TO SYS014-UT-3410-S-FILE3.)
000120     SELECT OUT-PUT ASSIGN TO SYS015-UT-3410-S.
000130 DATA DIVISION.
000140 FILE SECTION.
000150 FD  IN-PUT-1
000160     RECORD CONTAINS 80 CHARACTERS
000170     BLOCK CONTAINS 3 RECORDS
000180     LABEL RECORD IS STANDARD.
000190 01  IN-REC1 PIC X(80).
000200 FD  IN-PUT-2
000210     RECORD CONTAINS 80 CHARACTERS
000220     BLOCK CONTAINS 3 RECORDS
000230     LABEL RECORD IS STANDARD.
000240 01  IN-REC2 PIC X(80).
000250 FD  IN-PUT-3
000260     RECORD CONTAINS 80 CHARACTERS
000270     BLOCK CONTAINS 3 RECORDS
000280     LABEL RECORD IS STANDARD.
000290 01  IN-REC3 PIC X(80).
000300 FD  OUT-PUT
000310     RECORD CONTAINS 80 CHARACTERS
000320     BLOCK CONTAINS 3 RECORDS
000330     LABEL RECORD IS OMITTED.
000340 01  OUT-REC.
000350     05  FILLER PIC X(4).
000360     05  CODA PIC X.
000370         88  HI VALUE "9".
000380     05  FILLER PIC X(6).
000390     05  CODB PIC X.
000400         88  LO VALUE "8".
000410     05  FILLER PIC X(68).
000420 PROCEDURE DIVISION.
000430     OPEN INPUT IN-PUT-1 OUTPUT OUT-PUT.
000440 IN-1.
000450     READ IN-PUT-1 INTO OUT-REC AT END GO TO CLOSE1.
000460 TESTER.
000470     IF HI AND LO PERFORM CLOSE-OUT ELSE WRITE OUT-REC.(3)
000480 A.  GO TO IN-1.
000490 CLOSE1.
000500     CLOSE IN-PUT-1 WITH NO REWIND.      )
000510     OPEN INPUT IN-PUT-2 WITH NO REWIND.} (2)
000520 IN-2.
000530     READ IN-PUT-2 INTO OUT-REC AT END GO TO CLOSE2.
000540     PERFORM TESTER.
000550     GO TO IN-2.
```

Figure 45. Creating an Unlabeled Multivolume File (Part 1 of 2)

```
000560 CLOSE2.
000570     CLOSE IN-PUT-2 WITH NO REWIND.      } (2)
000580     OPEN INPUT IN-PUT-3 WITH NO REWIND. }
000590 IN-3.
000600     READ IN-PUT-3 INTO OUT-REC AT END GO TO CLOSE3.
000610     PERFORM TESTER.
000620     GO TO IN-3.
000630 CLOSE-OUT.
000640     CLOSE OUT-PUT REEL.                                    } (3)
000650     STOP "REMOVE TAPE ON SYS015 AND MOUNT NEW TAPE".       }
000660 CLOSE3.
000670     CLOSE IN-PUT-3 OUT-PUT. (4)
000680     STOP RUN.


// LBLTYP TAPE (5)
// EXEC LNKEDT


// ASSGN SYS014,X'283'
// TLBL FILE1,'MULTI-FILE1 VOL' }
// TLBL FILE2,'MULTI-FILE2 VOL' } (6)
// TLBL FILE3,'MULTI-FILE3 VOL' }
// ASSGN SYS015,X'282' (7)
// EXEC
```

Figure 45. Creating an Unlabeled Multivolume File (Part 2 of 2)

MASS STORAGE FILE LABELS

The IBM Disk Operating System/Virtual Storage provides postive identification and protection of all files on mass storage devices by recording labels on each volume. These labels ensure that the correct volume is used for input, and that no current information is destroyed on output.

The mass storage labels always include one volume label for each volume and one or more file labels for each logical file on the volume. There may also be user header labels and user trailer labels.

Volume Labels

The volume label is an 80-byte data field preceded by a 4-byte key field. Both the key field and the first four bytes of the data field contain the label identifier VOL1. IOCS creates a standard volume label for every volume processed by the Disk Operating System/Virtual Storage. It is always the third record on cylinder 0, track 0. The format and contents of a standard volume label can be found in the publication DOS/VS Disk Labels.

Standard File Labels

A standard file label identifies a particular logical file, gives its location(s) on the mass storage device, and contains information to prevent premature destruction of current files. A standard file label for a file located on a mass storage device is a 140-character label created (OPEN/CLOSE OUTPUT) in part by IOCS using the DLBL control statement. The fields contained within the label follow three standard formats.

1. Format 1 is used for all logical files. The contents of the fields of a Format 1 label is discussed in "Appendix C: Standard Mass Storage Device Labels."

2. Format 2 is required for indexed files. The contents of the fields of a Format 2 label can be found in the publication DOS/VS Disk Labels.

3. Format 3 is required if a logical file uses more than three extents of any volume. The contents of the fields of a Format 3 label can be found in the publication DOS/VS Disk Labels.

User Labels

The programmer can include additional labels to further define his file. The labels are referred to as user standard labels. They cannot be specified for indexed files. A user label is an 80-character label containing UHL (user header label) or UTL (user trailer label) in the first three character positions. The fourth position contains a number 1 through 8 which represents the relative position of the user label with a group of user labels. The contents of the remaining 76 positions is entirely up to the programmer. User header and trailer labels are written on the first track of the first extent of each volume allocated by the programmer for the file. User header labels are resequenced starting with one (UHL1) at the beginning of each new volume.

LABEL PROCESSING CONSIDERATIONS

Files on Mass Storage Device Opened as Input

1. Standard labels checked

   a. The volume serial numbers in the volume labels are compared to the file serial numbers in the EXTENT card.

   b. Fields 1 through 3 in Format 1 label are compared to the corresponding fields in the DLBL card.

   c. Each of the extent definitions in the Format 1 and Format 3 labels is checked against the limit fields supplied in the EXTENT card.

2. User labels checked

   a. If user header labels are indicated for directly or sequentially organized files, they are read as each volume of the file is opened. After reading each label, the OPEN routine branches to the programmer's label routine if the appropriate USE AFTER STANDARD LABEL PROCEDURE declarative is specified in the source program. The LABEL RECORDS clause with the data-name option must be specified in the Data Division. The programmer's label routine then performs any processing required.

b. If user trailer labels are indicated on a sequential file, they are read after reaching the end of the last extent on each volume when the file is closed, provided end-of-file has been reached. Trailer labels are processed by the programmer's label routine if the appropriate USE AFTER STANDARD LABEL PROCEDURE declarative is specified in the source program. The LABEL RECORDS clause with the data-name option must be specified in the Data Division.

Files on Mass Storage Devices Opened as Output

1. Standard labels created

   a. The volume serial numbers in the volume labels are compared to the file serial numbers in the EXTENT card.

   b. The extent definitions in all current labels on the volume are checked to determine whether any extend into those defined in the EXTENT card. If any overlap, the expiration date is checked against the current date in the Communication Region of the Supervisor. If the expiration date has passed, the old labels are deleted. If not, the operator is notified of the condition.

   c. The new Format 1 label is written with information supplied in the DLBL card. If an indexed file is being processed, the DTFIS routine supplies information for the Format 2 label.

   d. The information in the EXTENT card is placed in the Format 1 labels and, if necessary, in the additional Format 3 labels.

2. User header labels created

   a. If user header labels are indicated by the presence of the appropriate USE AFTER STANDARD LABEL PROCEDURE declarative and the LABEL RECORDS clause with the data-name option, the programmer's label routine is entered to furnish the labels as each volume of the file is opened. This can be done for as many as eight user header labels per volume. As each label is presented, IOCS writes it out on the first track of the first extent of the volume.

   b. If user trailer labels are indicated by the presence of the appropriate USE AFTER STANDARD LABEL PROCEDURE declarative and the LABEL RECORDS clause with the data-name option, the programmer's label routine is entered to furnish the labels when the end of the last extent on each volume is reached. This can be done for as many as eight user trailer labels. The CLOSE statement must be issued to create trailer labels for the last volume of a sequential file or for a direct file.

UNLABELED FILES

When a multivolume tape file is opened as INPUT and integer as specified in the ASSIGN clause is greater than 1, the compiler will generate the following message to the operator:

```
C126D IS IT EOF?
```

The operator must respond either with N if it is not the last reel, or with Y if it is the last reel. If it is end-of-file, control passes to the imperative-statement specified in the AT END phrase of the READ statement; if it is not end-of-file, processing of the next volume is initiated.

If the integer specified in the ASSIGN clause is not greater than 1, control always passes at end-of-volume to the imperative-statement specified in the AT END phrase of the READ statement.

The IBM DOS/VS COBOL Compiler and Library support the American National Standard Code for Information Interchange (ASCII) as well as EBCDIC. This support allows the user at object time to accept and create magnetic tapes in accordance with all of the following standards:

- American National Standard Code for Information Interchange, X3.4-1968.
- American National Standard Magnetic Tape Labels for Information Interchange, X3.27-1969.
- American National Standard Recorded Magnetic Tape for Information Interchange (800 CPI, NRZI), X3.22-1967.

COBOL LANGUAGE CONSIDERATIONS

The ASCII feature is supported by the following addition to IBM's implementation of American National Standard COBOL:

The system-name specified in the ASSIGN clause is now coded as

SYSnnn-UT-device-C[-buffer offset][-name]

where

Organization code C indicates that an ASCII-encoded tape file is to be processed.

Buffer offset is a two-character field that serves to indicate the size of the block prefix. A block prefix, if present, precedes each physical record and is not accessible to the COBOL programmer. This entry may only be present for ASCII tape files and is only required if a non-zero block prefix exists. For output files, buffer offset may be specified as 00 for F, U, or D-mode records, or as 04 for D-mode records only. A buffer offset of 04 on output means that the block prefix will contain the length of each physical record. For input files, buffer offset may be in the range 00 through 99.

FILE HANDLING

In processing ASCII files, the supported record formats are fixed, undefined, and variable. A variable-length record on an ASCII file is known as a D-format record. ASCII support does not extend to spanned records. Record formats are discussed in detail in the chapter "Record Formats."

For an ASCII file that contains a buffer offset field, the following considerations apply:

- If the BLOCK CONTAINS clause with the RECORDS option is specified, or if the BLOCK CONTAINS clause is omitted, the compiler compensates for the buffer offset field.
- If the BLOCK CONTAINS clause with the CHARACTERS option is specified, the programmer must include the buffer offset as part of the physical record.

Labels on ASCII files are processed as are the existing DOS/VS standard and user standard labels.

Nonstandard label procedures, however, are not supported. Therefore, USE BEFORE STANDARD LABEL PROCEDUREs are not permitted for ASCII files. ASCII files on unlabeled tapes are supported. These unlabeled tapes may contain data in any of the supported record formats. A complete discussion of tape file labels can be found in the chapter "Advanced Processing Capabilities."

The ASCII option (organization code C in the ASSIGN clause) must not be specified for a file on which checkpoints are to be written.

Diagnostic messages associated with ASCII file handling are provided. At compile time, E-level messages are issued for files whose record descriptions contain data formats that are inconsistent with ASCII conversion. At object time, a message is issued if an invalid sign configuration is present during translation, and the job will be terminated.

OPERATIONAL CONSIDERATIONS

It should be noted that ASCII support causes translation from ASCII to EBCDIC on input and from EBCDIC to ASCII on output. Translation occurs automatically and is transparent to the COBOL programmer. Since an ASCII file is assumed to contain only ASCII characters, standard character substitution occurs when untranslatable configurations are present. The character X'1A' is substituted for invalid EBCDIC configurations during translation. An invalid ASCII configuration (high-order bit on) translates to the character X'3F'.

OBTAINING AN ASCII COLLATING SEQUENCE ON A SORT

If an ASCII collated sort is desired or numeric sort keys contain a sign in the form of a leading overpunch or separate character, a Program Product IBM DOS/VS Sort/Merge program must be used. If sort files reside on a 3330 or 3340 device, the Sort program that supports these devices is required. The Program Product IBM DOS/VS Tape and Disk Sort/Merge, Program Number 5746-SM1 is designed specifically for use with a DOS/VS system.

To obtain an ASCII collated sort, the system-name in the ASSIGN clause for the sort work files should contain a C in the _organization_ field. The _class_ field may be specified as either UT or DA. (Since ASCII support causes translation from ASCII to EBCDIC on input, sort work files are not restricted to tapes.)

Note that for an ASCII collated sort, the buffer offset field is not permitted.

The ASCII collating sequence is listed in the publication _IBM DOS Full American National Standard COBOL_.

Logical records for files which are not VSAM files may be in one of four formats: fixed-length (format F), variable-length (format V), undefined (format U), or spanned (format S). All of these formats are not supported for all access methods. F-mode files must contain records of equal lengths. Files containing records of unequal lengths must be V-mode, S-mode, or U-mode. Files containing logical records that are longer than physical records must be S-mode.

The record format is specified in the RECORDING MODE clause in the Data Division. If this clause is omitted, the compiler determines the record format from the record descriptions associated with the file. If the file is to be blocked, the BLOCK CONTAINS clause must be specified in the Data Division.

The prime consideration in the selection of a record format is the nature of the file itself. The programmer knows the type of input his program will receive and the type of output it will produce. The selection of a record format is based on this knowledge as well as an understanding of the type of input/output devices on which the file is written and of the access method used to read or write the file.

Coding considerations for non-fixed length records are discussed in the chapter "Table Handling Considerations."

FIXED-LENGTH (FORMAT F) RECORDS

Format F records are fixed-length records. The programmer specifies format F records by including RECORDING MODE IS F in the file description entry in the Data Division. If the clause is omitted and both of the following are true:

- All records in the file are the same size
- BLOCK CONTAINS [integer-1 TO] integer-2... does not specify integer-2 less than the length of the maximum level-01 record

the compiler determines the recording mode to be F. All records in the file are the same size if there is only one record description associated with the file and it contains no OCCURS clause with the DEPENDING ON option, or if multiple record descriptions are all the same length.

The number of logical records within a block (blocking factor) is normally constant for every block in the file. When fixed-length records are blocked, the programmer specifies the BLOCK CONTAINS clause in the file description entry in the Data Division.

In unblocked format F, the logical record constitutes the block. The BLOCK CONTAINS clause is unnecessary for unblocked records.

Format F records are shown in Figure 46. The optional control character, represented by C, is used for stacker selection and carriage control. When carriage control or stacker selection is desired, the WRITE statement with the ADVANCING or POSITIONING option is used to write records on the output file. In this case one character position must be included as the first character of the record. This position will be automatically filled in with the carriage control or stacker select character. The carriage control character never appears when the file is written on the printer or punched on the card punch.

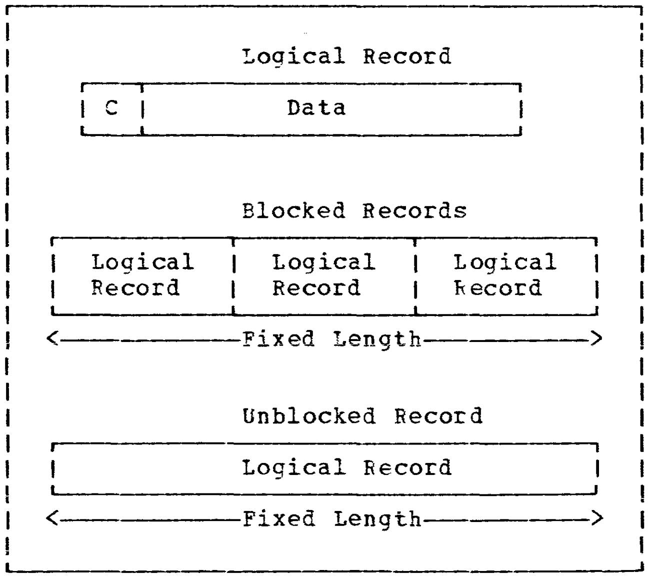


Figure 46. Fixed-Length (Format F) Records

UNDEFINED (FORMAT U) RECORDS

Format U is provided to permit the processing of any blocks that do not conform to F or V formats. Format U records are shown in Figure 47. The optional control character C, as discussed under "Fixed-Length (Format F) Records," may be used in each logical record.

The programmer specifies format U records by including RECORDING MODE IS U in the file description entry in the Data Division. U-mode records may be specified only for directly organized or standard sequential files.

If the RECORDING MODE clause is omitted, and BLOCK CONTAINS [integer-1 TO] integer-2... does not specify integer-2 less than the maximum level-01 record, the compiler determines the recording mode to be U if the file is directly organized and one of the following conditions exist:

- The FD entry contains two or more level-01 descriptions of different lengths.
- A record description contains an OCCURS clause with the DEPENDING ON option.
- A RECORD CONTAINS clause specifies a range of record lengths.

Each block on the external storage media is treated as a logical record. There are no record-length or block-length fields.

Note: When a READ INTO statement is used for a U-mode file, the size of the longest record for that file is used in the MOVE statement. All other rules of the MOVE statement apply.

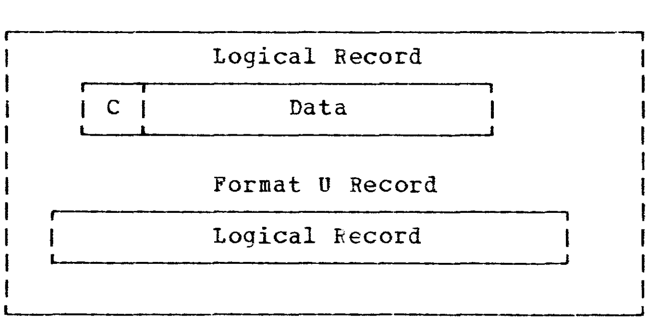


Figure 47. Undefined (Format U) Records

VARIABLE-LENGTH RECORDS

There are two types of variable-length record: D-format and V-format. A D-format record is a variable-length record on an ASCII tape file. D-format records are processed in the same manner as V-format records on tape files.

The programmer specifies format V records by including RECORDING MODE IS V in the file description entry in the Data Division. V-mode records may only be specified for standard sequential files. If the RECORDING MODE clause is omitted and BLOCK CONTAINS [integer-1 TO] integer-2... does not specify integer-2 less than the maximum level-01 record, the compiler determines the recording mode to be V if the file is standard sequential and one of the following conditions exists:

- The FD entry contains two or more level 01 descriptions of different lengths.
- A record description contains an OCCURS clause with the DEPENDING ON option.
- A RECORD CONTAINS clause specifies a range of record lengths.

V-mode records, unlike U-mode or F-mode records, are preceded by fields containing control information. These control fields are illustrated in Figures 48 and 49.

The first four bytes of each block contain control information (CC):

LL -- represents two bytes designating the length of the block (including the 'CC' field).

BB -- represents two bytes reserved for system use.

The first four bytes of each logical record contain control information (cc):

ll -- represents two bytes designating the logical record length (including the 'cc' field).

bb -- represents two bytes reserved for system use.

For unblocked V mode records (see Figure 45) the data portion + CC + cc constitute the block.

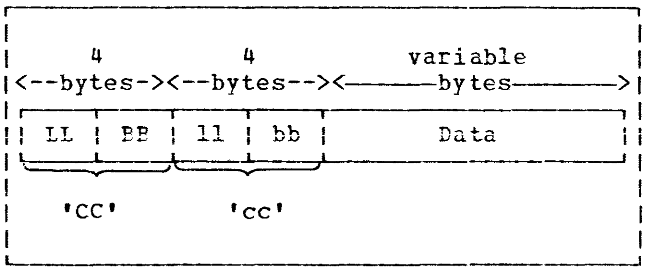


Figure 48. Unblocked V-Mode Records

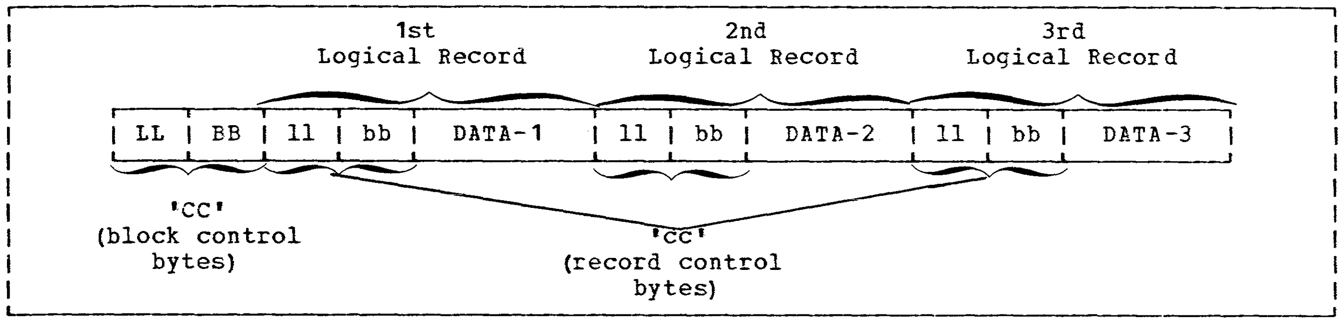


Figure 49. Blocked V-Mode Records

For blocked V-mode records (see Figure 49) the data portion of each record + the cc of each record + CC constitute the block.

The control bytes are automatically provided when the file is written and are not communicated to the programmer when the file is read. Although they do not appear in the description of the logical record provided by the programmer, the compiler will allocate input and output buffers which are large enough to accomodate them. When variable-length records are written on unit record devices, control bytes are neither printed nor punched. They appear, however, on other external storage devices as well as in buffer areas of storage. V-mode records moved from an input buffer to a working-storage area will be moved without the control bytes.

Note: When a READ INTO statement is used for a V-mode file, the size of the longest record for that file is used in the MOVE statement. All other rules of the MOVE statement apply.

Example 1:

Consider the following standard sequential file consisting of unblocked V-mode records:

```
FD  VARIABLE-FILE-1
    RECORDING MODE IS V
    BLOCK CONTAINS 35 TO 80 CHARACTERS
    RECORD CONTAINS 27 TO 72 CHARACTERS
    DATA RECORD IS VARIABLE-RECORD-1
    LABEL RECORDS ARE STANDARD.

01  VARIABLE-RECORD-1.
    05  FIELD-A     PIC X(20).
    05  FIELD-B     PIC 99.
    05  FIELD-C  OCCURS 1 TO 10 TIMES
        DEPENDING ON
        FIELD-B     PIC 9(5).
```

The LABEL RECORDS clause is always required. The DATA RECORD(S) clause is never required. If the RECORDING MODE clause is omitted, the compiler determines the mode as V since the record associated with VARIABLE-FILE-1 varies in length depending on the contents of FIELD-B. The RECORD CONTAINS clause is never required. The compiler determines record sizes from the record description entries. Record length calculations are affected by the following:

- When the BLOCK CONTAINS clause with the RECORDS option is used, the compiler adds four bytes to the logical record length and four more bytes to the block length.

- When the BLOCK CONTAINS clause with the CHARACTERS option is used, the programmer must include each cc + CC in the length calculation (see Figure 49). In the definition of VARIABLE-FILE-1, the BLOCK CONTAINS clause specifies 8 more bytes than does the record contains clause. Four of these bytes are the logical record control bytes and the other four are the block control bytes.

Assumming that FIELD-B contains the value 02 for the first record of a file and FIELD-B contains the value 03 for the second record of the file, the first two records will appear on an external storage device and in buffer areas of storage as shown in Figure 50.

If the file described in Example 1 had a blocking factor of 2, the first two records would appear on an external storage medium as shown in Figure 51.

```
              1st Block                                      2nd Block
|0040|BB|0036|bb|FIELD-A|02|FIELD-C|FIELD-C|0045|BB|0041|bb|FIELD-A|03|FIELD-C|FIELD-C|FIELD-C|

Note: Lengths appear in decimal notation for illustrative purposes.
```

Figure 50. Fields in Unblocked V-Mode Records

```
                       1st Record                                2nd Record
|0081|BB|0036|bb|FIELD-A|02|FIELD-C|FIELD-C|0041|bb|FIELD-A|03|FIELD-C|FIELD-C|FIELD-C|

Note: Lengths appear in decimal notation for illustrative purposes.
```

Figure 51. Fields in Blocked V-Mode Records

Example 2:

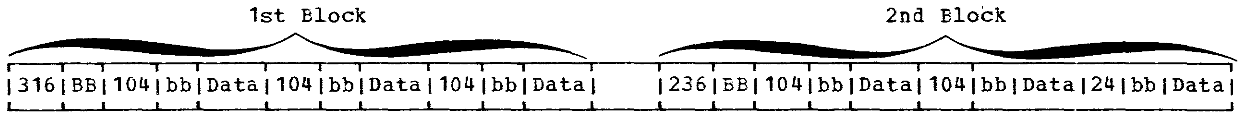
If VARIABLE-FILE-2 is blocked, with space allocated for three records of maximum size per block, the following FD entry could be used when the file is created:

```
FD  VARIABLE-FILE-2
    RECORDING MODE IS V
    BLOCK CONTAINS 3 RECORDS
    RECORD CONTAINS 20 TO 100 CHARACTERS
    DATA RECORDS ARE VARIABLE-RECORD-1,
    VARIABLE-RECORD-2
    LABEL RECORDS ARE STANDARD.

01  VARIABLE-RECORD-1.
    05  FIELD-A   PIC X(20).
    05  FIELD-B   PIC X(80).

01  VARIABLE-RECORD-2.
    05  FIELD-X   PIC X(20).
```

As mentioned previously, the RECORDING MODE, RECORD CONTAINS, and DATA RECORDS clauses are unnecessary. By specifying that each block contains three records, the programmer allows the compiler to provide space for three records of maximum size plus additional space for the required control bytes. Hence, 316 character positions are reserved by the compiler for each output buffer. If this size is other than the maximum, the BLOCK CONTAINS clause with the CHARACTERS option should be specified.

Assuming that the first six records written are five 100-character records followed by one 20-character record, the first two blocks of VARIABLE-FILE-2 will appear on the external storage device as shown in Figure 52.

The buffer for the second block is truncated after the sixth WRITE statement is executed since there is not enough space left for a maximum size record. Hence, even if the seventh WRITE to VARIABLE-FILE-2 is a 20-character record, it will appear as the first record in the third block. This situation can be avoided by using the APPLY WRITE-ONLY clause when creating files of variable-length blocked records.


1st Block
2nd Block
|316|BB|104|bb|Data|104|bb|Data|104|bb|Data|
|236|BB|104|bb|Data|104|bb|Data|24|bb|Data|

Note: Lengths appear in decimal notation for illustrative purposes.

Figure 52. First Two Blocks of VARIABLE-FILE-2

APPLY WRITE-ONLY Clause

The APPLY WRITE-ONLY clause is used to make optimum use of buffer and external storage space when creating a standard sequential file with blocked V-mode records.

Suppose VARIABLE-FILE-2 is being created with the following FD entry:

```
FD  VARIABLE-FILE-2
    RECORDING MODE IS V
    BLOCK CONTAINS 316 CHARACTERS
    RECORD CONTAINS 20 TO 100 CHARACTERS
    DATA RECORDS ARE VARIABLE-RECORD-1,
    VARIABLE-RECORD-2
    LABEL RECORDS ARE STANDARD.

01  VARIABLE-RECORD-1.
    05  FIELD-A  PIC X(20).
    05  FIELD-B  PIC X(80).

01  VARIABLE-RECORD-2.
    05  FIELD-X  PIC X(20).
```

The first three WRITE statements to the file create one 20-character record followed by two 100-character records. Without the APPLY WRITE-ONLY clause, the buffer is truncated after the third WRITE statement is executed, since the maximum size record no longer fits. The block is written as shown below:

```
|236|bb|24|bb|Data|104|bb|Data|104|bb|Data|
```

Using the APPLY WRITE-ONLY clause will cause a buffer to be truncated only when the next record does not fit in the buffer. That is, if the next three WRITE statements to the file specify VARIABLE-RECORD-2, the block will be created containing six logical records, as shown below:

```
|308|bb|24|bb|Data|104|bb|Data|104|bb|Data ≶

        ≶ 24|bb|Data|24|bb|Data|24|bb|Data|
```

Note: When using the APPLY WRITE-ONLY clause, records must not be constructed in buffer areas. An intermediate work area must be used with a WRITE FROM statement.

If an APPLY WRITE-ONLY clause is specified for a file and an OCCURS DEPENDING ON clause is specified within a record description of the file, the object of the OCCURS DEPENDING ON clause should not be defined within the record description for the file.

SPANNED (FORMAT S) RECORDS

A spanned record is a logical record that may be contained in one or more physical blocks. Format S records may be specified for direct files and for standard sequential files assigned to magnetic tape or to mass storage devices.

When creating files with S-mode records, if a record is larger than the remaining space in a block, a segment of the record is written to fill the block. The remainder of the record is stored in the next block or blocks, as required.

When retrieving a file with S-mode records, only complete records are made available to the programmer.

Spanned records are preceded by fields containing control information. Figure 53 illustrates the control fields.

BDF (Block Descriptor Field):

LL -- represents 2 bytes designating the length of the physical block (including the block descriptor field itself).

BB -- represents 2 bytes reserved for system use.

SDF (Segment Descriptor Field):

ll -- represents 2 bytes designating the length of the record segment (including the segment descriptor field itself).

bb -- represents 2 bytes reserved for system use.

Note: There is only one block descriptor field at the beginning of each physical block. There is, however, one segment descriptor field for each record segment within the block.

Each segment of a record in a block, even if it is the entire record, is preceded by a segment descriptor field. The segment descriptor field also indicates whether the segment is the first, the last, or an intermediate segment. Each block includes a block descriptor field. These fields are not described in the Data Division; provision is automatically made for them. These fields are not available to the programmer.

A spanned blocked file may be described as a file composed of physical blocks of fixed length established by the programmer. The logical records may be either fixed or variable in length and that size may be smaller, equal to, or larger than the physical block size. There are no required relationships between logical records and physical block sizes.

A spanned unblocked file may be described as a file composed of physical blocks each containing one logical record or one segment of a logical record. The logical records may be either fixed or variable in length. When the physical block contains one logical record, the length of the block is determined by the logical record size. When a logical record has to be segmented, the system always writes the largest physical block possible. The system segments the logical record when the entire logical record cannot fit on the track.

Figure 54 is an illustration of blocked spanned records of SFILE. SFILE is described in the Data Division with the following file description entry:

```
FD SFILE
   RECORD CONTAINS 250 CHARACTERS
   BLOCK CONTAINS 100 CHARACTERS
   .
   .
   .
```

Figure 54 also illustrates the concept of record segments. Note that the third block contains the last 50 bytes of REC-1 and the first 50 bytes of REC-2. Such portions of logical records are called record segments. It is therefore correct to say that the third block contains the last segment of REC-1 and the first segment of REC-2. The first block contains the first segment of REC-1 and the second block contains an intermediate segment of REC-1.

S-MODE CAPABILITIES

Formatting a file in the S-mode allows the programmer to make the most efficient use of external storage while organizing data files with logical record lengths most suited to his needs.

1. Physical record lengths can be designated in such a manner as to make the most efficient use of track capacities on mass storage devices.

2. The programmer is not required to adjust logical record lengths to maximum physical record lengths and their device-dependent variants when designing his data files.

3. The programmer has greater flexibility in transferring logical records across DASD types.

Spanned record processing will support the 2400, 3410, 3420 tape series, the 2311, 2314, 2319, 3330, and 3340 disk storage devices, and the 2321 data cell drive.

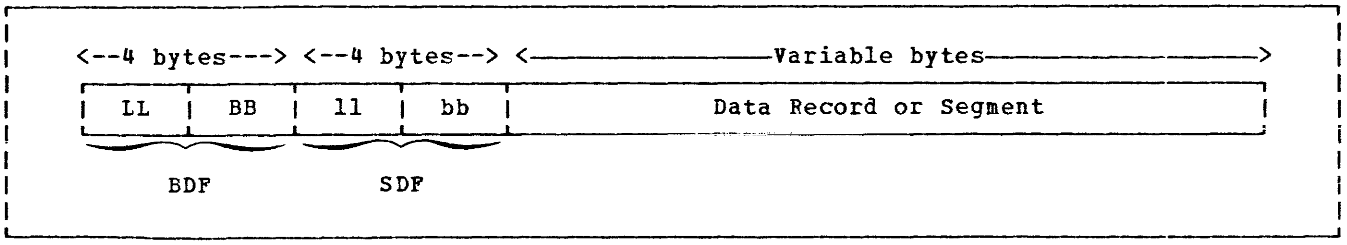


Figure 53. Control Fields of an S-Mode Record

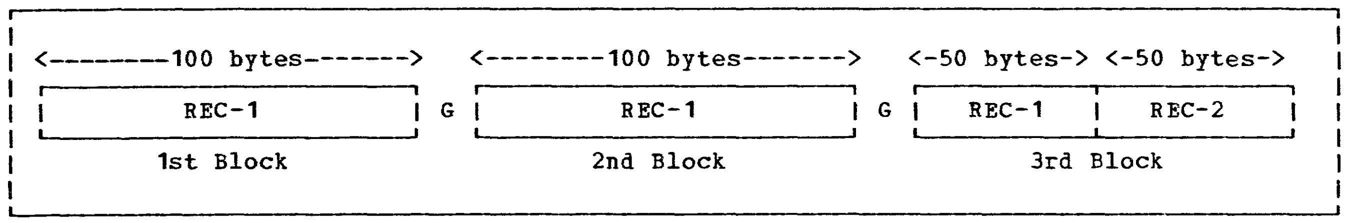


Figure 54. One Logical Record Spanning Physical Blocks

SEQUENTIALLY ORGANIZED S-MODE FILES ON TAPE OR MASS STORAGE DEVICES

When the spanned format is used for DTFMT or DTFSD files, the logical records may be either fixed or variable in length and are completely independent of physical record length. A logical record may span physical records. A physical record may contain one or more logical records and/or segments of logical records.

Source Language Considerations

The programmer specifies S-mode by describing the file with the following clauses in the file description (FD) entry of his COBOL program:

- BLOCK CONTAINS integer-2 CHARACTERS
- RECORD CONTAINS [integer-1 TO] integer-2 CHARACTERS
- RECORDING MODE IS S

The size of the physical record must be specified using the BLOCK CONTAINS clause with the CHARACTERS option. Any block size may be specified. Block size is independent of logical record size.

The size of the logical record may be specified by the RECORD CONTAINS clause. If this clause is omitted, the compiler will determine the maximum record size from the record descriptions under the FD.

Format S may be specified by the RECORDING MODE IS S clause. If this clause is omitted, the compiler will set the recording mode to S if the BLOCK CONTAINS integer-2 CHARACTERS clause was specified and either:

1. integer-2 is less than the largest fixed-length level-01 FD entry
2. integer-2 is less than the maximum length of a variable level-01 FD entry (i.e., an entry containing one or more OCCURS clauses with the DEPENDING ON option).

When the spanned recording mode is being used, each logical record is processed in a work area, not in the buffer. Logical records are always aligned on a double-word boundary. Therefore, the programmer is not required to add inter-record slack bytes for alignment purposes.

Except for the APPLY WRITE-ONLY clause, all the options for a variable file apply to a spanned file.

Processing Sequentially Organized S-Mode Files

Suppose a file has the following file description entry:

```
FD  SPAN-FILE
    BLOCK CONTAINS 100 CHARACTERS
    LABEL RECORDS ARE STANDARD
    DATA RECORD IS DATAREC.


01  DATAREC.
    05 FIELD-A PIC X(100).
    05 FIELD-B PIC X(50).
```

Figure 55 illustrates the first four blocks of SPAN-FILE as they would appear on external storage devices (i.e., tape or mass storage) or in buffer areas of virtual storage.

Note:

1. The RECORDING MODE clause is not specified. The compiler determines the recording mode to be S since the block size is less than the record size.
2. The length of each physical block is 100 bytes, as specified in the BLOCK CONTAINS clause. All required control fields, as well as data, must be contained within these 100 bytes.
3. No provision is made for the control fields within the level-01 entry DATAREC.

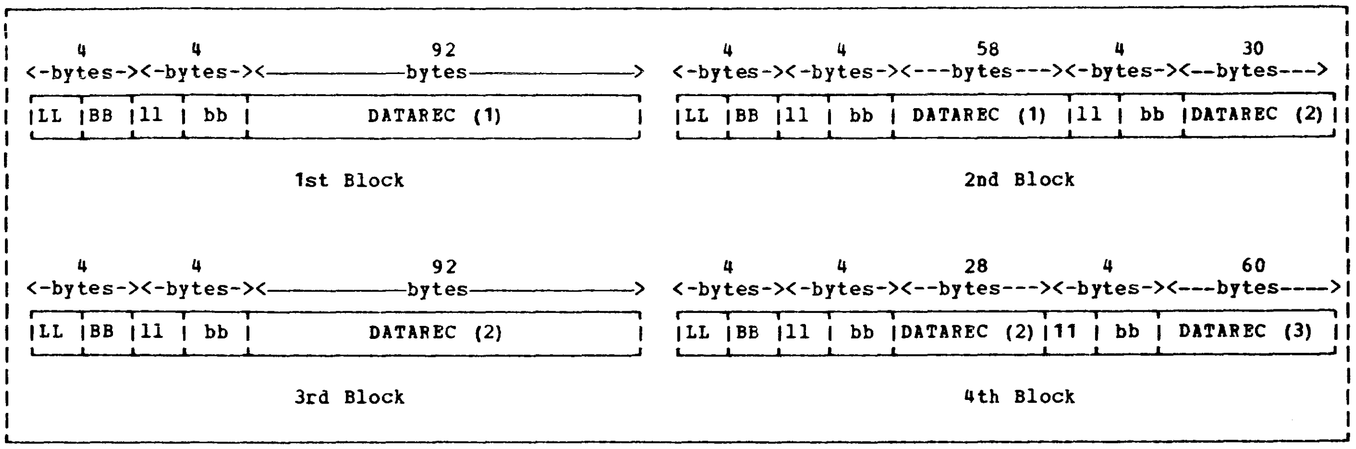
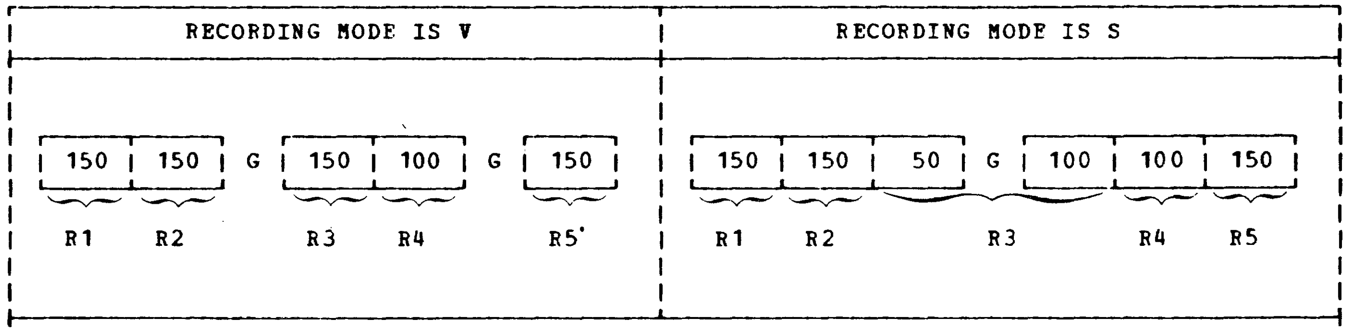


Figure 55. First Four Blocks of SPAN-FILE



Note: The enclosed diagrams are for illustrative purposes only. Neither takes into account the space required for control fields.

Figure 56. Advantage of S-Mode Records Over V-Mode Records

The preceding discussion dealt with S-mode records which were larger than the physical blocks that contained them. It is also possible to have S-mode records which are equal to or smaller than the physical blocks that contain them. In such cases, the RECORDING MODE clause must specify S (if so desired) since the compiler cannot determine this by comparing block size and record size.

One advantage of S-mode records over V-mode records is illustrated by a file with the following characteristics:

1. RECORD CONTAINS 50 TO 150 CHARACTERS
2. BLOCK CONTAINS 350 CHARACTERS
3. The first five records written are 150, 150, 150, 100, and 150 characters in length.

For V-mode records, buffers are truncated if the next logical record is too large to be completely contained in the block (see Figure 56). This results in more physical blocks and more inter-record gaps on the external storage device.

Note: For V-mode records, buffer truncation occurs:

1. When the maximum level-01 record is too large.
2. If APPLY WRITE-ONLY or SAME RECORD AREA is specified and the actual logical record is too large.

For S-mode records, all blocks are 350 bytes long and records that are too large to fit entirely into a block will be segmented. This results in more efficient use of external storage devices since the number of inter-record gaps are minimized (Figure 56).

With the exception of the last block, the actual physical block size will always fall between the limits of specified block size and four bytes less than the specified block size, depending on whether or not the residual space of an incomplete block in the buffer is sufficient to add a segment length field and at least one byte of data. That is, specified block size - 4 ≤ actual block size ≤ specified block size.

The last block may be short when an incomplete block remains in the buffer at CLOSE time.

A second advantage of S-mode processing over that of V-mode is that the programmer is no longer limited to a record length that does not exceed the track capacity of the mass storage device selected. Records may span track, cylinders, and extents, but not volumes.

DTFMT and DTFSD spanned records differ from other formats because of an allocation of an area of storage known as the "logical record area." If logical records span physical blocks, COBOL will use this logical record area to assemble complete logical records. If logical records do not span blocks (i.e., they are contained within a single physical block) the logical record area is not used. Regardless, it is complete logical records that are made available to the programmer. Both READ and WRITE statements should be thought of as manipulating complete logical records and not record segments.

DIRECTLY ORGANIZED S-MODE FILES

When S-mode is used for a directly organized file, only unblocked records are permitted. Logical records may be either fixed or variable in length. A logical record will span physical records if, and only if, it spans tracks. A physical record will contain only one logical record or a segment of a logical record, or segments of two logical records and/or whole logical records. Records may span tracks, cylinders, and extents, but not volumes.

Source Language Considerations

The programmer specifies S-mode by describing the file with the following clauses in the file description (FD) entry of his COBOL program:

- BLOCK CONTAINS integer-2 CHARACTERS
- RECORD CONTAINS [integer-1 TO] integer-2 CHARACTERS
- RECORDING MODE IS S

The size of a logical record may be specified by the RECORD CONTAINS clause. If this clause is omitted, the compiler will determine the maximum record size from the record descriptions under the FD.

The spanned format may be specified by the RECORDING MODE IS S clause. If this clause is omitted, the compiler will set the recording mode to S if the BLOCK CONTAINS integer-2 CHARACTERS clause was

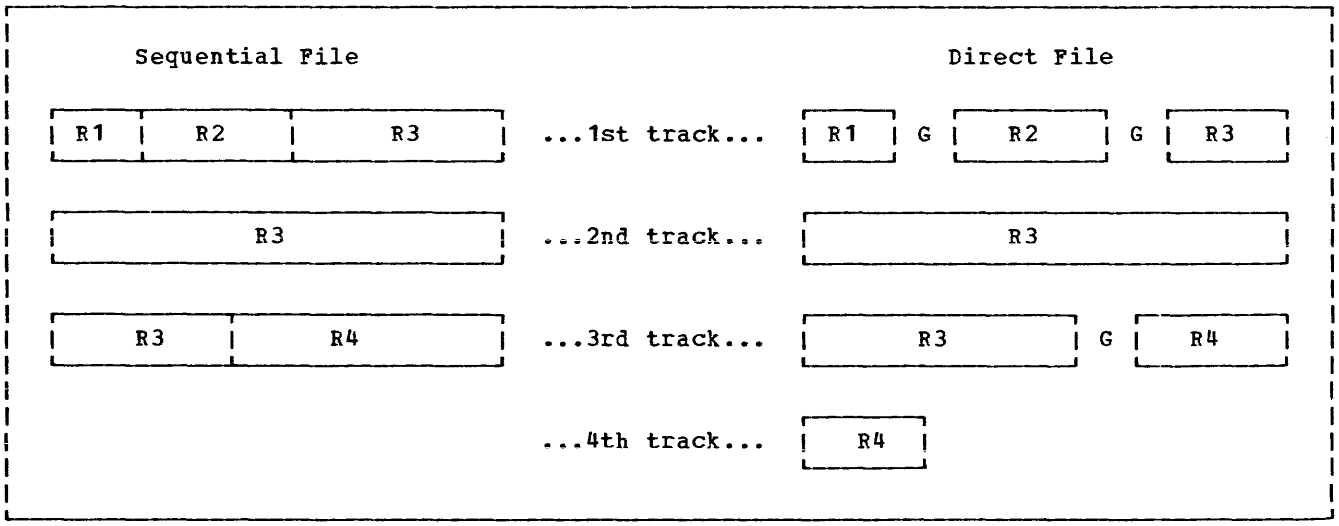


Figure 57. Direct and Sequential Spanned Files on a Mass Storage Device

specified and integer-2 is less than the greatest logical record size. This is the only use of the BLOCK CONTAINS clause. It is otherwise treated as comments.

The physical block size is determined by either:

1. The logical record length, or

2. The track capacity of the device being used.

If, for example, the track capacity of a mass storage device is 3625 characters, any record smaller than 3625 characters may be written as a single physical block. If a logical record is greater than 3625 characters, the record is segmented. The first segment may be contained in a physical block of up to 3625 bytes, and the remaining segments must be contained in succeeding blocks. In other words, a logical record will span physical blocks if, any only if, it spans tracks.

Figure 57 illustrates four variable-length records (R1, R2, R3, and R4) as they would appear in direct and sequential files on a mass storage device. In both cases, control fields have been omitted for illustrative purposes. For both files, assume:

1. BLOCK CONTAINS 3625 CHARACTERS (track capacity = 3625)

2. RECORD CONTAINS 500 TO 5000 CHARACTERS

In the sequential file, each physical block is 3625 bytes in length and is completely filled with logical records. The file consists of three physical blocks, occupies three tracks, and contains no inter-record gaps.

In the direct file, the physical blocks vary in length. Each block contains only one logical record or one record segment. Logical record R3 spans physical blocks only because it spans tracks. The file consists of seven physical blocks, occupies more than three tracks, and contains three inter-record gaps.

Processing Directly Organized S-Mode Files

When processing directly organized files, there are two advantages spanned format has over the other record formats:

1. Logical record lengths may exceed the length restriction of the track capacity of the mass storage device. If, for example, the track capacity of a mass storage device is 2000 bytes, the length of each logical record for formats other than spanned is, by necessity, restricted to the track capacity.

   Note: Even when the spanned format is used, the COBOL restriction on the length of logical records (i.e., a maximum length of 32,767 characters) must be adhered to.

2. For formats other than spanned, only complete logical records can be written on any single track. This means that if a track has only 1000 unoccupied bytes and the programmer attempts to add a record of 1100 bytes to this track, an INVALID KEY condition will occur. When the spanned format is used, a 1000 byte segment will be written on the specified track, and the remainder will be written on the next track. The segmenting is transparent to the programmer.

PART III

This chapter describes techniques and hints for better COBOL programming.

CODING CONSIDERATIONS FOR DOS/VS

These suggestions will aid DOS/VS efficiency:

- If a short subprogram is referenced only once or twice (and is not an exception condition routine), then its code should be incorporated in the calling program, if convenient.
- Subprograms and frequently used subroutines should be loaded near the programs which use them. This can be done via linkage editor control cards.
- Segmentation in many cases is no longer necessary or desirable.
- Data items of constant value should be grouped together. Data items whose values vary during execution should also be grouped together and should be separate from those of constant value, if feasible.
- FDs for files that will be opened at the same time should be grouped together.
- The most frequently referenced data items should be placed in the beginning of the Working Storage Section.
- The COBOL Procedure Division should be organized generally as follows:
  - All frequently used paragraphs or sections should be located near the routines that use them.
  - All infrequently used paragraphs or sections should be grouped together and apart from frequently used routines. The COUNT option can be used as an aid in this process.
- Avoid initializing data areas until just before they are needed.
- Reference data in the order in which it is stored.
- Use the OPTIMIZE feature if possible.

  Note: When OPT is in effect, the generated code is more suitable for running under VS, as the addressing scheme is designed to reduce possible page faults.

  Further, the procedure is divided into 4K blocks, each of which is assigned a PBL. Since these blocks correspond to two pages each, the user may get some idea of the inter-page relationships in his program (although the first is not page aligned). The statement range for each PBL is given on the compiler output listing. This should help the user rearrange his program if he so desires.
- The REDEFINES clause should be used for its alternate grouping and alternate description capabilities rather than for merely saving space. Although it will save virtual space, it can lead to coding errors if not used carefully.

GENERAL CONSIDERATIONS

COPY

The COPY function should be used by an installation so that if a record format, for example, changes, each program does not need to be modified itself. Rather, the COPY library is updated and each program then recompiled.

Use of this function can lead to standardization of naming conventions and ease of maintenance.

SYNTAX CHECKING

The first several compilations of a program should use the CSYNTAX or SYNTAX feature to save compilation time.

Formatting the Source Program Listing

The lister feature increases significantly the usability of the source program listing, not only by producing cross-reference information, but by formatting the listing to aid logic tracing. There are four statements that can be coded in any or all of the four divisions of a source program: SKIP1, SKIP2, SKIP3, and EJECT. These statements provide the programmer with the ability to control the spacing of a source listing and thereby improve its readability. These statements should not be used when the lister feature is used.

ENVIRONMENT DIVISION

RESERVE Clause

When using an additional buffer to process standard sequential or indexed files, care must be taken to ensure that the buffer is filled before the execution of each WRITE or REWRITE statement.

APPLY WRITE-ONLY Clause

To make optimum use of buffer and external storage space allocated when creating a standard sequential file with blocked V-mode records, the programmer should use the APPLY WRITE-ONLY clause for the file. Using this clause causes a buffer to be truncated only when the next record does not fit in the buffer. (If APPLY WRITE-ONLY is not specified, the buffer is truncated when the maximum size record will not fit in the space remaining in the buffer.)

DATA DIVISION

STORAGE CONSIDERATIONS

The amount of storage used for all FD entries, the WORKING STORAGE SECTION, and REPORT SECTION must not exceed 1 Mb, since the compiler can only handle a maximum of 255 for BL-CELLS. One BL-CELL is assigned for each file or for 4096, whichever comes first.

OVERALL CONSIDERATIONS

FD Entries

File Description (FD) entries for the most active files should appear first, since the COBOL compiler assigns registers to files until it runs out of registers, and then reuses the last registers for all subsequent files. This does not apply when OPT is in effect, since in that case the compiler will determine the frequency of usage and assign registers accordingly.

Prefixes

Assign a prefix to each level-01 item in a program, and use this prefix on every subordinate item (except FILLER) to associate a file with its records and work areas. For example, MASTER is the prefix used here:

```
FILE SECTION.
FD  MASTER-INPUT-FILE
     .
     .
     .
01  MASTER-INPUT-RECORD.
     .
     .
     .
WORKING-STORAGE SECTION.
01  MASTER-WORK-AREA.
    05  MASTER-PAYROLL PICTURE 9(3).
    05  MASTER-SSNO PICTURE 9(9).
```

If files or work areas have the same fields, use the prefix to distinguish between them. For example, if three files all have a date field, instead of DATE, DAT, and DA-TE, use MASTER-DATE, DETAIL-DATE, and REPORT-DATE. Using a unique prefix for each level-01 item and all subordinate fields makes it easier for a programmer unfamiliar with the program to find fields in the program listing, and to know which fields are logically part of the same record or area.

When using the MOVE statement with the CORRESPONDING option and referring to individual fields, redefine or rename "corresponding" names with the prefixed unique names. This technique eliminates excessive qualifying. For example:

```
01  MST-WORK-AREA.
    05  SAME-NAMES.      (***)
        10  LAST-NAME  PIC...
        10  FIRST-NAME  PIC...
        10  PAYROLL  PIC...
        .
        .
        .
    05  DIFF-NAMES REDEFINES SAME-NAMES.
        10  MST-LAST-NAME  PIC...
        10  MST-FIRST-NAME  PIC...
        10  MST-PAYROLL  PIC...
01  RPT-WORK-AREA.
    05  SAME-NAMES.      (***)
        10  PAYROLL  PIC...
        10  FILLER  PIC...
        10  FIRST-NAME  PIC...
        10  FILLER  PIC...
        10  LAST-NAME  PIC...
        .
        .
        .
PROCEDURE DIVISION.
    .
    .
    IF MST-PAYROLL IS EQUAL TO HDQ-PAYROLL
    AND MST-LAST-NAME
    IS NOT EQUAL TO PRRV-LAST-NAME
    MOVE CORRESPONDING
    MST-WORK-AREA
    TO RPT-WORK-AREA.
```

Note: Fields marked *** above must have exactly the same names for their subordinate fields to be considered "corresponding." The same names must not be the redefining ones or they will not be considered to correspond.

Level Numbers

The programmer should use widely incremented level numbers such as 01, 05, 10, 15, etc., instead of 01, 02, 03, 04, etc., in order to allow space for future insertions of group levels. For readability, indent level numbers. (The lister feature does this automatically, even if the original source program does not follow such indenting practices.)

Note that when using the SYMDMP option, level numbers appear "normalized" in the symbolic dump produced. For example, a group of data items described as:

```
01 RECORDA.
   05 FIELD-A.
      10 FIELD-A1 PIC X.
      10 FIELD-A2 PIC X.
```

will appear as follows in SYMDMP output:

```
   01 RECORDA...
   02 FIELD-A...
   03 FIELD-A1...
   03 FIELD-A2...
```

Use level number 88 for codes. Thus, if the codes must be changed, the Procedure Division coding for tests need not be changed.

FILE SECTION

RECORD CONTAINS Clause

The programmer should use the RECORD CONTAINS clause with the integer CHARACTERS option in order to save himself, as well as any future programmer, the task of counting the data record description positions. In addition, the compiler can then diagnose errors if the data record description conflicts with the RECORD CONTAINS clause.

BLOCK CONTAINS Clause

If a block prefix exists on an ASCII file and the BLOCK CONTAINS clause is used in the COBOL program, the length of the block prefix must be included in the BLOCK CONTAINS clause.

WORKING-STORAGE SECTION

Separate Modules

In a large program, the programmer may wish to plan ahead for breaking the programs into separately compiled modules, as follows:

1. When using separate modules, an attempt should be made to combine entries of each Working-Storage Section into a single level-01 record (or a single level-01 record for each 32K bytes). Logical record areas can be indicated by using level-02, -03, etc., entries. A CALL statement with the USING option is more efficient when a single item is passed than when many level-01 and/or -77 items are passed. When this method is employed, mistakes are more easily avoided.

2. Areas which do not contain VALUE clauses should be separated from areas that do contain VALUE clauses. VALUE clauses (except for level-88 items) are invalid in the Linkage Section.

3. When the Working-Storage Section consists of one level-01 item without any VALUE clauses, the COPY statement can easily be used to include the item as the description of a Linkage Section in a separately compiled module.

4. See the chapter "Using the Segmentation Feature" for information on how to modularize the Procedure Division of a COBOL program; VS coding considerations should also be taken into account.

Locating the Working-Storage Section in Dumps

If the SYMDMP option is not used for program debugging, a method of locating the Working-Storage Section of a program in object-time dumps is to include the two following statements as the first and last Working-Storage statements, respectively, in the program.

```
77  FILLER PICTURE X(44), VALUE "PROGRAM
    XXXXXXXX WORKING-STORAGE BEGINS HERE".

01  FILLER PICTURE X(42), VALUE "PROGRAM
    XXXXXXXX WORKING-STORAGE ENDS HERE".
```

These two nonnumeric literals will appear in all dumps of the program, delimiting the Working-Storage Section. The program-name specified in the PROGRAM-ID clause should replace the XXXXXXXX in the literal.

The location and length of Working-Storage is given in the compiler output when SYM, LISTX, or CLIST is in effect.

REDEFINES Clause

REUSING DATA AREAS: Virtual storage can be used more efficiently by writing different data descriptions for the same data area. For example, the coding that follows shows how the same area can be used as a work area for the records of several input files that are not processed concurrently. Caution should be exercised when using this procedure, as it can lead to programming errors.

```
WORKING-STORAGE SECTION.
01  WORK-AREA-FILE1.
     (largest record description for FILE1)
.
.
.
01  WORK-AREA-FILE2 REDEFINES
    WORK-AREA-FILE1.
     (largest record description for FILE2)
.
.
.
```

ALTERNATE GROUPINGS AND DESCRIPTIONS: Program data can often be described more efficiently by providing alternate groupings or data descriptions for the same data. For example, a program references both a field and its subfields, each of which is more efficiently described with a different usage. This can be done by using the REDEFINES clause as follows:

```
01  PAYROLL-RECORD.
05  EMPLOYEE-RECORD PICTURE X(28).
05  EMPLOYEE-FIELD REDEFINES
    EMPLOYEE-RECORD.
    10  NAME PICTURE X(24).
    10  NUMBERX PICTURE S9(5) COMP.
05  DATE-RECORD PICTURE X(10).
```

The following illustrates how a table (TABLEA) can be initialized by having different data descriptions for the same data:

```
05  VALUE-A.
    10  A1 PICTURE S9(9) COMPUTATIONAL
        VALUE IS ZEROES.
    10  A2 PICTURE S9(9) COMPUTATIONAL
        VALUE IS 1.
    .
    .
    .
    10  A100 PICTURE S9(9) COMPUTATIONAL
        VALUE IS 99.
05  TABLEA REDEFINES VALUE-A
    PICTURE S9(9) COMPUTATIONAL
    OCCURS 100 TIMES.
```

Note: Caution should be exercised when redefining a subscript. If the value of the redefining data item is changed in the Procedure Division, a new calculation for the subscript is performed only if a new paragraph is entered.

PICTURE Clause

DECIMAL-POINT ALIGNMENT: Procedure Division operations are most efficient when the decimal positions of the data items involved are aligned. If they are not, the compiler generates instructions to align the decimal positions before any operations involving the data items can be executed. This is referred to as scaling.

Assume, for example, that a program contains the following instructions:

```
WORKING-STORAGE SECTION.
77  A PICTURE S999V99.
77  B PICTURE S99V9.
.
.
.
PROCEDURE DIVISION.
.
.
.
   ADD A TO B.
```

Time and internal storage space are saved by defining B as:

```
77 B PICTURE S99V99.
```

If it is inefficient to define B differently, a one-time conversion can be done, as explained in "Data Format Conversion" in this chapter.

FIELDS OF UNEQUAL LENGTH: When a data item is moved to another data item of a different length, the following should be considered:

- If the items are external decimal items, the compiler generates instructions to insert zeros in the high-order positions of the receiving field, when it is the larger.
- If the items are nonnumeric, the compiler may generate instructions to insert spaces in the low-order positions of the receiving field (or the high-order positions if the JUSTIFIED RIGHT clause is specified). This generation of extra instructions can be avoided if the sending field is described with a length equal to or greater than the receiving field.

SIGN USAGE: The presence or absence of a plus or minus sign in the description of an arithmetic field often can affect the efficiency of a program. The following paragraphs discuss some of the considerations.

Decimal Items: The sign position in an internal or external decimal item can contain:

1. A plus or minus sign. If S is specified in the PICTURE clause, a plus or minus sign is inserted when either of the following conditions prevail:

   a. The item is in the Working-Storage Section and a VALUE clause has been specified.

   b. A value for the item is assigned as a result of an arithmetic operation during execution of the program.

   If an external decimal item is punched, printed, or displayed, an overpunch will appear in the low-order digit. In EBCDIC, the configuration for low-order zeros normally is a nonprintable character. Low-order digits of positive values will be represented by one of the letters A through I (digits 1 through 9); low-order digits of negative values will be represented by one of the letters J through R (digits 1 through 9).

2. A hexadecimal F. If S is not specified in the PICTURE clause, an F is inserted in the sign position when either of the following conditions prevail:

   a. The item is in the Working-Storage Section and a VALUE clause has been specified

   b. A value for the item is developed during the execution of the program.

   An F is treated as positive, but is not an overpunch.

3. An invalid configuration. If an internal or external decimal item contains an invalid configuration in the sign position, and if the item is involved in a Procedure Division operation, the program will be abnormally terminated.

Note: If the SIGN clause is used and it specifies that the sign is LEADING, more object code will be generated when that data item is used with a verb. The additional code is needed to move the sign character to the TRAILING position before performing the operation.

Unsigned items (items for which no S has been specified) are treated as absolute values. Whenever a value (signed or unsigned) is stored in or moved in an elementary move to an unsigned item, a hexadecimal F is stored in the sign position of the unsigned item. For example, if an arithmetic operation involves signed operands and an unsigned result field, compiler-generated code will insert an F in the sign position of the result field when the result is stored.

For internal and external decimal items used as input, it is the programmer's responsibility to ensure that the input data is valid. The compiler does not generate a test to ensure that the configuration in the sign position is valid.

When a group item is being moved, the data is moved without regard to the level structure of the group items involved. The possibility exists that the configuration in the sign position of a subordinate numeric item may be destroyed. Therefore, caution should be exercised in moving group items with subordinate numeric fields or with other group operations such as READ or ACCEPT.

USAGE Clause

DATA FORMAT CONVERSION: Operations involving mixed, elementary numeric data formats require conversion to a common format. This usually means that additional storage is used and execution time is increased. The code generated must often move data to an internal work area, perform any necessary conversion, and then execute the indicated operation. Often, too, the result may have to be converted in the same way. Table 31 indicates when data conversion is necessary.

If it is impractical to use the same data formats throughout a program, and if two data items of different formats are frequently used together, a one-time conversion can be effected. For example, if A is defined as a COMPUTATIONAL item and B as a COMPUTATIONAL-3 item, A can be moved to a work area that has been defined as COMPUTATIONAL-3. This move causes the data in A to be converted to COMPUTATIONAL-3. Whenever A and B are used in a Procedure Division operation, reference can be made to the work area rather than to A. When this technique is used, the conversion is performed only once, instead of each time an operation is performed.

Table 31. Data Format Conversion (Part 1 of 2)

Usage	Bytes Required	Boundary Alignment Required	Typical Usage	Converted for Arithmetic Operations	Special Characteristics
DISPLAY (external decimal)	1 per digit (except for V)	No	Input from cards, output to cards, listings	Yes	May be used for numeric fields up to 18 digits long. Fields over 15 digits require extra instructions if used in computations.
DISPLAY (external floating point)	1 per character (except for V)	No	Input from cards, output to cards, listings	Yes	Converted to COMP-2 format via COBOL library subroutine.
COMP-3 (internal decimal)	1 per 2 digits plus 1 byte for low-order digit and sign	No	Input to a report item Arithmetic fields Work areas	Sometimes when a small COMP-3 item is used with a small COMP item	Requires less space than DISPLAY. Convenient form for decimal alignment. Can be used in arithmetic computations without conversion. Fields over 15 digits require a subroutine when used in computations.
COMP (binary)	2 if $1 \leq N \leq 4$ 4 if $5 \leq N \leq 9$ 8 if $10 \leq N \leq 18$ where N is the number of 9's in the picture	Halfword Fullword Fullword	Subscripting Arithmetic fields	Sometimes for both mixed and unmixed usages	Rounding and testing for the ON SIZE ERROR condition are cumbersome if calculated result is greater than 9(9). Extra instructions are generated for computations if the SYNCHRONIZED clause is not specified. Fields of over nine digits require additional handling.

Table 31. Data Format Conversion (Part 2 of 2)

Usage	Bytes Required	Bountary Alignment Required	Typical Usage	Converted for Arithmetic Operations	Special Characteristics
COMP-1 (internal floating point)	4 (short-precision)	Fullword	Fractional exponentiation	No	Tends to produce less accurate results if more than 17 significant digits are required and if the exponent is large. Extra instructions are generated for computations if the SYNCHRONIZED clause is not specified. Requires floating-point feature.
COMP-2 (internal floating point)	8 (long-precision)	Double-word	Fractional exponentiation when additional precision is required	No	Same as COMP-1.

The following seven cases show how data conversions are handled on mixed elementary items for names, data comparisions, and arithmetic operations. Moves without the CORRESPONDING option to and from group items, as well as comparisons involving group items, are done without conversion.

Numeric DISPLAY to COMPUTATIONAL-3:

To Move Data: Converts DISPLAY data to COMPUTATIONAL-3 data.

To Compare Data: Converts DISPLAY data to COMPUTATIONAL-3 data.

To Perform Arithmetic Operations: Converts DISPLAY data to COMPUTATIONAL-3 data.

Numeric DISPLAY to COMPUTATIONAL:

To Move Data: Converts DISPLAY data to COMPUTATIONAL-3 data and then to COMPUTATIONAL data.

To Compare Data: Converts DISPLAY to COMPUTATIONAL or converts both DISPLAY and COMPUTATIONAL data to COMPUTATIONAL-3 data.

To Perform Arithmetic Operations: Converts DISPLAY data to COMPUTATIONAL-3 or COMPUTATIONAL data.

COMPUTATIONAL-3 to COMPUTATIONAL:

To Move Data: Moves COMPUTATIONAL-3 data to a work area and then converts COMPUTATIONAL-3 data to COMPUTATIONAL data.

To Compare Data: Converts COMPUTATIONAL data to COMPUTATIONAL-3 or vice versa, depending on the size of the field.

To Perform Arithmetic Operations: Converts COMPUTATIONAL data to COMPUTATIONAL-3 or vice versa, depending on the size of the field.

COMPUTATIONAL to COMPUTATIONAL-3:

To Move Data: Converts COMPUTATIONAL data to COMPUTATIONAL-3 data in a work area, and then moves the work area.

To Compare Data: Converts COMPUTATIONAL to COMPUTATIONAL-3 data or vice versa, depending on the size of the field.

To Perform Arithmetic Operations: Converts COMPUTATIONAL to COMPUTATIONAL-3 data or vice versa, depending on the size of the field.

COMPUTATIONAL to Numeric DISPLAY:

To Move Data: Converts COMPUTATIONAL data to COMPUTATIONAL-3 data and then to DISPLAY data.

To Compare Data: Converts DISPLAY to COMPUTATIONAL or both COMPUTATIONAL and DISPLAY data to COMPUTATIONAL-3 data, depending on the size of the field.

To Perform Arithmetic Operations: Depending on the size of the field, converts DISPLAY data to COMPUTATIONAL data, or both DISPLAY and COMPUTATIONAL data to COMPUTATIONAL-3 data in which case the result is generated in a COMPUTATIONAL-3 work area and then converted and moved to the DISPLAY result field.

COMPUTATIONAL-3 to Numeric DISPLAY:

To Move Data: Converts COMPUTATIONAL-3 data to DISPLAY data.

To Compare Data: Converts DISPLAY data to COMPUTATIONAL-3 data. The result is generated in a COMPUTATIONAL-3 work area and is then converted and moved to the DISPLAY result field.

Numeric DISPLAY to Numeric DISPLAY:

To Perform Arithmetic Operations: Converts all DISPLAY data to COMPUTATIONAL-3 data. The result is generated in a COMPUTATIONAL-3 work area and is then converted to DISPLAY and moved to the DISPLAY result field.

Internal Floating-point to Any Other: When an item described as COMPUTATIONAL-1 or COMPUTATIONAL-2 (internal floating-point) is used in an operation with another data format, the item in the other data format is always converted to internal floating-point. If necessary, the internal floating-point result is then converted to the format of the other data item.

SYNCHRONIZED Clause

As illustrated in Table 31, COMPUTATIONAL, COMPUTATIONAL-1 and COMPUTATIONAL-2 items have specific boundary alignment requirements. To ensure correct alignment, either the programmer or the compiler may have to insert slack bytes or the compiler must generate extra instructions to move the item to a correctly aligned work area when reference is made to the item.

The SYNCHRONIZED clause may be used at the elementary level to specify the automatic alignment of elementary items on their proper boundaries, or at the 01 level to synchronize all elementary items within the group. For COMPUTATIONAL items, if the PICTURE is in the range of S9 through S9(4), the item is aligned on a halfword boundary. If the PICTURE is in the range of S9(5) through S9(18), the item is aligned on a fullword boundary. For COMPUTATIONAL-1 items, the item is aligned on a fullword boundary. For COMPUTATIONAL-2 items, the item is aligned on a doubleword boundary. The SYNCHRONIZED clause and slack bytes are fully discussed in the publication IBM System/360 Disk Operating System: Full American National Standard COBOL.

Special Considerations for DISPLAY and COMPUTATIONAL Fields

NUMERIC DISPLAY FIELDS: Zeros are not inserted into numeric DISPLAY fields by the instruction set. When numeric DISPLAY data is moved, the compiler generates instructions that insert any necessary zeros into the DISPLAY fields. When numeric DISPLAY data is compared, and one field is smaller than the other, the compiler generates instructions to move the smaller item to a work area where zeros are inserted.

COMPUTATIONAL FIELDS: COMPUTATIONAL fields can be aligned on either a halfword or fullword boundary. If an operation involves COMPUTATIONAL fields of different lengths, the halfword field is automatically expanded to a fullword field. Therefore, mixed halfword and fullword fields require no additional operations.

COMPUTATIONAL-1 AND COMPUTATIONAL-2 FIELDS: If an arithmetic operation involves a mixture of short-precision and long-precision fields, the compiler generates instructions to expand the short-precision field to a long-precision field before the operation is executed.

COMPUTATIONAL-3 FIELDS: The compiler does not have to generate instructions to insert high-order zeros for ADD and SUBTRACT statements that involve COMPUTATIONAL-3 data. The zeros are inserted by the instruction set.

Data Formats in the Computer

The following examples illustrate how the various COBOL data formats appear in the computer in EBCDIC (Extended

Binary-Coded-Decimal Interchange Code) format. More detailed information about these data formats appear in the publication IBM System/370 Principles of Operation.

Numeric DISPLAY (External Decimal): Suppose the value of an item is -1234, and its PICTURE and USAGE clauses are:

```
     PICTURE 9999 DISPLAY.

             or

     PICTURE S9999 DISPLAY.
```

The item appears in the computer in the following forms, respectively:

$$| F1 | F2 | F3 | \underbrace{F4}_{\text{Byte}} |$$

$$| F1 | F2 | F3 | \underbrace{D4}_{\text{Byte}} |$$

Hexadecimal F is treated arithmetically as positive; hexadecimal D represents a minus sign.

COMPUTATIONAL-3 (Internal Decimal): Suppose the value of an item is +1234, and its PICTURE and USAGE clauses are:

```
     PICTURE 9999 COMPUTATIONAL-3.

             or

     PICTURE S9999 COMPUTATIONAL-3.
```

The item appears internally in the following forms, respectively:

$$| 01 | 23 | \underbrace{4F}_{\text{Byte}} |$$

$$| 01 | 23 | \underbrace{4C}_{\text{Byte}} |$$

Hexadecimal F is treated arithmetically as positive; hexadecimal C represents a plus sign.

Note: Since the low-order byte of an internal decimal number always contains a sign field, an item with an odd number of digits can be stored more efficiently than an item with an even number of digits.

Note that a leading zero is inserted in the above example.

COMPUTATIONAL (Binary): Suppose the value of an item is 1234, and its PICTURE and USAGE clauses are:

```
     PICTURE S9999 COMPUTATIONAL.
```

The item appears internally in the following form:

```
        | 0000 | 0100 | 1101 | 0010 |
        └──────┴──────┴──────┴──────┘
          ↑
          |
        Sign
    Position
```

A 0 in the sign position indicates that the number is positive. Negative numbers are represented in two's complement form; thus, the sign position of a negative number will always contain a 1.

For example -1234 would appear as follows:

```
        | 1111 | 1011 | 0010 | 1110 |
        └──────┴──────┴──────┴──────┘
          ↑
          |
        Sign
    Position
```

Binary Item Manipulation: A binary item is allocated storage ranging from one halfword to two fullwords, depending on the number of 9's in its PICTURE. Table 32 is an illustration of how the compiler allocates this storage. Note that it is possible for a value larger than that implied by the PICTURE clause to be stored in the item. For example, PICTURE S9(4) implies a maximum value of 9,999, although it could actually hold the number 32,767.

Because most binary items are manipulated according to their allotted storage capacity, the programmer can ignore this situation. For the following reasons, however, he must be careful of his data:

1. When the ON SIZE ERROR option is used, the size test is made on the basis of the maximum value allowed by the picture of the result field. If a size error condition exists, the value of the result field is not altered and control is given to the imperative-statements specified by the error option.

Table 32. Relationship of PICTURE to Storage Allocation

PICTURE	Maximum Working Value	Assigned Storage
S9 through S9(4)	32,767	One halfword
S9(5) through S9(9)	2,147,483,647	One fullword
S9(10) through S9(18)	9,223,372,036,854,775,807	Two fullwords

Note: If TRUNC option is used and data is moved to decimal receiving field, then maximum working value for S9(10) through S9(18) PICTURE is 2,147,483,647,999,999,999.

2. When a binary item is displayed or exhibited, the value used is a function of the number of 9's specified in the PICTURE clause.

3. When the actual value of a positive number is significantly larger than its picture value, a value of 1 could appear in the sign position of the item, causing the item to be treated as a negative number in subsequent operations.

Figure 58 illustrates three binary manipulations. In each case, the result field is an item described as PICTURE S9 COMPUTATIONAL. One halfword of storage has been allocated, and no ON SIZE ERROR option is involved. Note that if the ON SIZE ERROR option had been specified, it would have been executed for cases B and C.

<u>COMPUTATIONAL-1 or COMPUTATIONAL-2 (Floating-point)</u>: Suppose the value of an item is +1234 and that its USAGE is COMPUTATIONAL-1, the item appears internally in the following form:

```
|0|100  0011|0100 1101 0010 0000 0000 0000|
 S 1       7 8                           31
```

S is the sign position of the number.

0 in the sign position indicates that the sign is plus.

1 in the sign position indicates that the sign is minus.

Bits 1 through 7 are the exponent (characteristic) of the number.

Bits 8 through 31 are the fraction (mantissa) of the number.

This form of data is referred to as floating point. The example illustrates short-precision floating-point data (COMPUTATIONAL-1). In long-precision (COMPUTATIONAL-2), the fraction length is 56 bits. (For a detailed explanation of floating-point representation, see the publication <u>IBM System/370 Principles of Operation</u>.)

PROCEDURE DIVISION

The Procedure Division of a program can often be made more efficient or easier to debug by using some of the techniques described below.

MODULARIZING THE PROCEDURE DIVISION

Modularization involves organizing the Procedure Division into at least three functional levels: a main-line routine, processing subroutines, and input/output subroutines. When the Procedure Division is modularized, programs are easier to maintain and document. In addition, modularization makes it simple to break down a program using the segmentation feature, resulting in a more efficient segmented program. Virtual storage implications should be taken into

Case	Hexadecimal Result of Binary Calculation	Decimal Equivalent	Actual Decimal Value in Halfword of Storage	DISPLAY or EXHIBIT Value
A	0008	8	+8	8
B	000A	10	+10	0
C	C350	50000	-15536	6

Figure 58. Treatment of Varying Values in a Data Item of PICTURE S9

consideration when rearranging the Procedure Division. The COUNT option is useful in determining a rearrangement scheme.

Main-Line Routine

The main-line routine should be short and simple, and should contain all the major logical decisions of the program. This routine controls the order in which second-level subroutines are executed. All second-level subroutines should be invoked from the main-line routine by PERFORM statements.

Processing Subroutines

Processing subroutines should be broken down into as many functional levels as necessary, depending on the complexity of the program. These must be completely closed subroutines, with one entry point and one exit point. The entry point should be the first statement of the subroutine. The exit point should be the EXIT statement. Processing subroutines can PERFORM only lower level subroutines; return to the higher level subroutine (processing subroutine) must be accomplished by a GO TO statement that references the EXIT statement.

Input/Output Subroutines

The input/output subroutines should be the lowest level subroutines, since all higher level subroutines have access to them. There should be one OPEN subroutine and one CLOSE subroutine for the program, and only one functional (READ or WRITE) subroutine for each file. Having one READ or WRITE subroutine per file has several advantages:

1. Coding can be added to count records on a file, transform blanks into zeros, check for 9's padding, etc.

2. Input and output files can be reformatted without changing the logic of the program.

3. DEBUG statements can be added during testing to create input or to DISPLAY formatted output, instead of having to create a test file.

OVERALL CONSIDERATIONS

OPTIMIZE Option

If the OPTIMIZE option is in effect, the number of procedure blocks in a program cannot exceed 255. A procedure block is equivalent to approximately 4096 bytes of Procedure Division code.

If the COUNT option is in effect, the number of verb blocks in a program cannot exceed 32,767. A verb block consists of a set of verbs in which any verb (excluding ABEND) in the block is executed if and only if all verbs in the block are executed. The average program Procedure Division contains approximately three verbs per verb block.

INTERMEDIATE RESULTS

The compiler treats arithmetic statements as a succession of operations and sets up intermediate result fields to contain the results of these operations. Examples of such statements are the arithmetic statements and statements containing arithmetic expressions. See the appendix "Intermediate Results" in the publication IBM DOS Full American National Standard COBOL for a description of the algorithms used by the compiler to determine the number of places reserved for intermediate result fields.

Intermediate Results and Binary Data Items

If an operation involving binary operands requires an intermediate result greater than 18 digits, the compiler converts the operands to internal decimal before performing the operation. If the result field is binary, the result will be converted from internal decimal to binary.

If an intermediate result will not be greater than nine digits, the operation is performed most efficiently on binary data fields.

Intermediate Results and COBOL Library Subroutines

If a decimal multiplication operation requires an intermediate result greater than 30 digits, a COBOL library subroutine is used to perform the multiplication. The result of this multiplication is then truncated to 30 digits.

A COBOL library subroutine is used to perform division if:

1. The divisor (scaled or not) is equal to or greater than 15 digits.

2. The length of the divisor (scaled or not) plus the length of the scaled dividend is greater than 16 bytes. The lengths of the operands are in decimal internally.

3. The <u>scaled dividend</u> is greater than 30 digits. (A scaled dividend is a number that has been multiplied by a power of ten in order to obtain the desired number of decimal places in the quotient.)

Intermediate Results Greater Than 30 Digits

Whenever the number of digits in a decimal intermediate result is greater than 30, the field is truncated to 30 digits. A warning message will be generated during compilation, and program flow will not be interrupted at execution time. This truncation may cause a result to be incorrect.

If binary or internal decimal data is in agreement with its data description, no interrupt can occur because of an overflow condition in an intermediate result. This is due to the truncation described in the preceding paragraph.

If the possibility exists that an intermediate result field may exceed 30 digits, truncation can be avoided by the specification of floating-point operands (COMPUTATIONAL-1 or COMPUTATIONAL-2); however, accuracy may not be maintained.

Intermediate Results and Floating-point Data Items

If a floating-point operand has an intermediate result field in which exponent overflow occurs, the job will be abnormally terminated.

Intermediate Results and the ON SIZE ERROR Option

The ON SIZE ERROR option applies only to the final calculated results and not to intermediate result fields.

EXPONENTIATION

When the exponent is not a literal, one of the following three subroutines is invoked, depending on the base and the exponent:

1. If the base is not a floating-point item and the exponent is an integer item, a call to the subroutine ILBDXPR0 is generated and the exponentiation is executed in packed decimal arithmetic.

2. If the base is a floating-point item and the exponent is an integer item, a call to the subroutine ILBDGPW0 is generated and the exponentiation is executed in floating-point arithmetic.

3. If the exponent is a floating-point item or has a PICTURE specifying decimal places, a call to the subroutine ILEDFPW0 is generated and the exponentiation is executed in floating-point arithmetic. The base is always treated as a positive number, regardless of sign and the answer will always be a positive number. Caution should therefore be exercised when using non-integer exponents.

When the exponent is an integer literal, one of the following applies:

1. If the base is a floating-point item, a call to the subroutine ILBDGPW0 is generated and the exponentiation is executed in floating-point arithmetic.

2. If the base is not a floating-point item, an in-line loop is generated to perform the exponentiation unless the maximum possible result exceeds 30 digits, in which case a call to the subroutine ILBDXPR0 is generated. In either case, the exponentiation is executed in packed decimal arithmetic.

Optimization Based on Execution Frequency

Additional optimization techniques may be used based on execution frequency statistics. These techniques are discussed in the chapter entitled "Execution Statistics".

COMPUTE Statement

The use of the COMPUTE statement generates more efficient code than does the use of individual arithmetic statements, since the compiler can keep track of internal work areas and does not have to store the results of intermediate calculations. It is the programmer's responsibility, however, to ensure that the data is defined with the level of significance required in the answer.

IF Statement

Nested and compound IF statements should be avoided as the logic is difficult to debug.

MOVE Statement

Performing a move operation for an item longer than 256 bytes requires the generation of more instructions than are required for a move operation for an item of 256 bytes or less.

For fields longer than 512 bytes, a MOVE LONG (MVCL) instruction is generated unless the first byte of the receiving field is used as a byte of the sending field. In this case, the object-time subroutine ILBDVMO0 is called to perform the move.

When a MOVE statement with the CORRESPONDING option is executed, data items are considered as "corresponding" only if their respective data-names are the same, including all implied qualification up to, but not including, the data-names used in the MOVE statement itself.

For example:

```
01 AA                    01  XX
   05 BB                     05 BB
     10 CC                     10 CC
     10 DD                     10 DD
   05 EE                     05 YY
     10 FF                     10 FF
```

The statement MOVE CORRESPONDING AA TO XX will result in moving CC, and DD, but not FF, since FF of EE does not correspond to FF of YY.

The compiler assumes that the data being moved conforms to PICTURE and USAGE specifications. If it does not, dissimilar results will occasionally occur because of the different code generated for various sending and receiving fields. This fact is most apparent when the sending field is COMPUTATIONAL, the value in the item exceeds the number of digits specified in the PICTURE clause, and the option NOTRUNC is in effect.

Note: The other rules for MOVE CORRESPONDING, of course, must still be satisfied.

NOTE Statement

When the NOTE statement is the first statement in a paragraph, it will cause the whole paragraph to be treated as part of the NOTE. Programmer errors can be avoided by using the asterisk (*) in place of the NOTE statement.

PERFORM Statement

PERFORM is a useful statement if the programmer adheres to the following rules:

1. Always execute the last statement of a series of routines being operated on by a PERFORM statement. When branching out of the routine, make sure control will eventually return to the last statement of the routine, which should be an EXIT statement. Although no code is generated, the EXIT statement allows a programmer to immediately recognize the extent of a series of routines within the range of a PERFORM statement.

2. Always either PERFORM routine-name THRU routine-name-exit, or PERFORM section-name. A PERFORM paragraph-name can create problems for the programmer trying to maintain the program. For example, if one paragraph must be broken into two paragraphs, the programmer must examine every statement to determine whether this paragraph is within the range of the PERFORM statement. As a result, all statements referencing the paragraph-name must be changed to PERFORM THRU statements.

3. A PERFORM statement containing embedded PERFORMs or PERFORM VARYING with one or more AFTER options causes the compiler to generate complex code. If a series of simple PERFORM statements can accomplish the same function, the programmer would be wise to substitute these since more efficient code is generated.

READ INTO AND WRITE FROM OPTIONS

Always use READ INTO and WRITE FROM, and process all files in the Working-Storage Section for the following reasons:

1. Debugging is much simpler. Working-Storage areas are easier to locate in a dump than are buffer areas. And, if files are blocked, it is much easier to determine which record in a block was being processed when the abnormal termination occurred.

2. Trying to access a record-area after the AT END condition has occurred (for example, AT END MOVE HIGH-VALUE TO INPUT-RECORD) can cause problems if the record area is defined only in the File Section.

Note: The programmer should be aware that additional time is used to execute the move operation involved in each READ INTO or WRITE FROM instruction.

TRANSFORM Statement

The TRANSFORM statement generates more efficient code than the EXAMINE REPLACING BY statement when only one character is being transformed. The TRANSFORM statement, however, uses a 256-byte table.

To use the Sort Feature, statements are written in the COBOL source program. These statements are described in IBM DOS Full American National Standard COBOL. The Sort/Merge publications listed in the Preface of this manual contain information on the Sort/Merge feature.

When a SORT or MERGE statement is used in a program, the compiler generates linkages between the program, modules in the subroutine library, and the Sort/Merge program. The name of Sort/Merge called by COBOL is "SORT" and the user must include the proper one on the option.

Depending on the features specified and devices to be used by Sort/Merge, different Sort/Merge products should be used:

Feature	Product Requirement
VSAM	5746-SM1
MERGE	5746-SM1
ASCII-Collated Sort	5743-SM1,5746-SM1
Numeric Sort keys with Sign in the form of leading overpunch or separate character.	5743-SM1,5746-SM1
3330/3333 Sort Work files	5743-SM1,5746-SM1
3330-11 Sort Work files	5746-SM1
3340 Sort Work files	5746-SM1
3350 Sort Work files	5746-SM1
3400 Sort Work files	5743-SM1,5746-SM1
2311 Sort Work files	SM-483,5743-SM1

The program product DOS/VS Sort/Merge, 5746-SM1, is designed specifically for use with DOS/VS.

Otherwise, IBM DOS Tape and Disk Sort/Merge, 360N-SM-483, can be used.

Additional job control statements must be included in the execution step of the job to describe the files used by the sort program. These statements are described below in "Sort Job Control Requirements."

Note: The Checkpoint/Restart Feature can be activated during a sorting operation by specifying the RERUN statement.

SORT/MERGE JOB CONTROL REQUIREMENTS

Three types of files can be defined for the Sort program in the execution job step: input, output, and work. Two types of files can be defined for the Merge program in the execution job step: input and output.

SORT INPUT AND OUTPUT CONTROL STATEMENTS

When the USING and/or GIVING options are specified, the compiler generates dummy Input and/or Output Procedures. Hence, the job control requirements for files named as operands of USING and GIVING are the same as those for files used as input to or output from the sorting operation in these procedures.

The following job control statements are required for files used as input to or output from the sorting operation:

```
ASSGN

  followed by

    VOL
    TPLAB

or

    VOL
    DLAB
    XTENT

or

    DLBL
    EXTENT

or

    TLBL
```

The symbolic unit to which each sort input or output file is assigned in the source language ASSIGN clause is specified in an ASSGN control statement.

Note: ASSGN control statements are required only if the input/output devices used in an application have not been previously assigned the appropriate symbolic names.

If an input file contains standard labels, a TLBL or DLBL (or VOL and TPLAB or VOL and DLAB) statement(s) is required. The symbolic name of the device from which the input file is to be read must also be included on this statement.

One EXTENT control statement is required to define the limits of each area of a mass storage device from which an input file will be read. EXTENT statements must include the symbolic unit name of the device containing the extent.

If the output file is to use standard labels, a TLBL or DLBL statement is required.

One EXTENT control statement must be used to define the limits of each area of a mass storage device onto which the output file is written. The symbolic name of the output unit must appear on this card.

Note: Because the USING and GIVING options generate dummy input and/or Output procedures, the rules on pooling of files in the Sort/Merge Programmer's Guide referenced above do not apply. No pooling of Sort input, output, and work files is allowed.

A SIZE parameter is needed in the EXEC statement when sorting a VSAM file. The SIZE parameter must be in the format SIZE=(AUTO,nK) to take into account the fetching of the sort module during execution and VSAM storage requirements.

SORT WORK FILE CONTROL STATEMENTS

The Sort program requires at least one mass storage unit or three tape units as an intermediate sort work file. The symbolic units to which this file is assigned are normally consecutively numbered beginning with SYS001. Intermediate storage may be assigned on the following devices:

- IBM 2400 Series Magnetic Tape Units
- IBM 3400 Series Magnetic Tape Units[1]
- IBM 2311 Direct-Access Storage Device
- IBM 2314/2319 Direct-Access Storage Facility
- IBM 3330/3333 Direct-Access Storage Facility[1]
- IBM 3330-11 Disk Storage[1]
- IBM 3340 Direct-Access Storage Facility[1]
- IBM 3350 Direct Access Storage[1]

Note: When variable-length or redefined-length records are being sorted, sort work files must not be assigned to 7-track tapes. 7-track tape work files can only be used to sort records whose keys are packed decimal or binary.

Device types may not be mixed; i.e., work units for a particular sort operation must all be of the same type

[1]Only supported by the DOS Sort/Merge Program Product, Program Number 5743-SM1 or the DOS/VS Sort/Merge Program Product, Program Number 5746-SM1 (see above).

If spanned records are being sorted and mass storage devices are being used as sort work files, it is the programmer's responsibility to assign these work files to devices whose track sizes are larger than the logical record sizes of the records being sorted. A spanned record that is larger than the available track size can be sorted by assigning the work files to magnetic tape.

If a work unit is to use standard labels, a TLBL or DLBL control statement is required. The filename entry on these statements must be SORTWK1 through SORTWKn. The symbolic unit names assigned to the work areas to be allocated (SYS001, SYS002, etc.) must appear on these cards.

One EXTENT control statement must be included to define each work area on a mass storage device. The total work area required may be divided into as many as eight extents, which would require eight EXTENT control statements. When code SD is specified on the DLBL card, symbolic unit names on these statements must be in consecutive order (SYS001, SYS002, etc.). If SORT-OPTION is specified, the symbolic unit names must be in the same order as specified on SORTWK.

If SYSLST is assigned to a disk, the option PRINT=NONE or ROUTE=LOG must be used.

Amount of Intermediate Storage Required

When intermediate storage is assigned on a mass storage unit, at least twice the amount required to hold all input records should be assigned. This area may consist of from one to eight extents, and the extents may be assigned on no more than eight devices.

If tape intermediate storage is used, at least the minimum number of units (three) must be assigned. The input file can be as large as the number of records that can be written on one full reel of tape. Assigning more than three intermediate storage tape drives does not increase the maximum input file size, but does improve performance.

Improving Performance

Performance increases significantly if 50K of real storage is available for execution of the Sort program. At the 100K level, the performance is very high. If insufficient virtual storage is available, the Sort/Merge program will issue a message:

```
7054A "INSUFFICIENT CORE"
```

SORT-OPTION Clause

The "SORT-OPTION" clause is a means of specifying the options that have been selected for the associated sort/merge operation that cannot be specified via the SORT special registers. The format of the contents of the data-name is shown in Figure 58.1. This corresponds to the SORT/MERGE option statement. No validity check is done by COBOL; only the address is passed to SORT/MERGE. For more details on specific options for SORT, see IBM DOS/VS Sort/Merge Programmer's Guide.

Note: The COBOL-SORT interface does not allow any preceding blanks in front of the "SORT-OPTION" clause. One and only one blank must follow the keyword OPTION. At least one blank must follow the last operand.

```
OPTION

        ┌PRINT          ┐
        │PRINT=NONE     │
        │PRINT=ALL      │
        └PRINT=CRITICAL ┘

        [,LABEL=(,,WORK)]

        ┌           ┌n         ┐ ┐
        │           │nK        │ │
        │,STORAGE=  │(n,VIRT)  │ │
        └           └(nK,VIRT) ┘ ┘

        [,ALTWK][,ERASE]

        ┌,ROUTE=LST┐
        └,ROUTE=LOG┘

        ┌,SORTWK=work               ┐
        └,SORTWK=(work1,...work)    ┘
```

Figure 58.1. OPTION Control Statement to SORT/MERGE

PRINT Option

```
┌PRINT          ┐
│PRINT=NONE     │
│PRINT=ALL      │
└PRINT=CRITICAL ┘
```

PRINT and PRINT=ALL specify that all messages are to be printed by the sort/merge program. This includes error and end-of-job messages, control card information, various size calculations, and other informative messages.

PRINT=NONE specifies that no messages are to be printed by the sort/merge program. This parameter is useful if you have no alternate message device and do not want messages listed with other printed output. A message device need not be assigned.

PRINT=CRITICAL specifies that only messages critical to the sort/merge program's operation are to be printed. These are error messages resulting from conditions that can cause program termination. For more details on these conditions and messages, refer to IBM DOS/VS Sort/Merge Programmer's Guide.

Note: PRINT=ALL is assumed until the OPTION statement is read, therefore, if PRINT=NONE or PRINT=CRITICAL will be used, the OPTION statement should precede all others.

LABEL Option

```
[LABEL=(,,work)]
```

This operand specifies the type of labels associated with the work files. The two label types are:

S - standard labels
U - unlabeled

The default is S, standard labels.

Work must be replaced by S or U. This operand is required if the OPTION statement is specified, and unlabeled work files are used. If the operand is omitted, standard labels are assumed for all files.

When standard labels are used, the sort/merge program uses the DOS/VS system facilities to process these labels. Unlabeled tape files are processed by the sort/merge program. No user programming is required.

STORAGE Option

```
┌           ┌n         ┐ ┐
│           │nK        │ │
│,STORAGE=  │(n,VIRT)  │ │
└           └(nK,VIRT) ┘ ┘
```

This option is required to specify to the sort/merge program how much storage to use and whether it can fix pages.

```
┌STORAGE=n        ┐
└STORAGE=(n,VIRT) ┘
```

n specifies the amount of storage to be made available to sort/merge (together with its user routines). n can be specified either as a decimal number of bytes, or as a decimal number of K (1024 bytes).

The default is the value of the SIZE parameter on the EXEC job control statement. If both SIZE and STORAGE are specified, the lower value is taken. If neither is specified, the default is the partition size or the required size calculated by sort/merge (but at least 64K), whichever is smaller. The sort/merge program terminates if n is less than 16K bytes. If n is greater than the partition size, it is ignored.

If the sort/merge program is invoked from another program, the defaults are calculated in a similar way, but the value of the SIZE parameter and the partition size are adjusted downwards by the difference between the address of the sort/merge load point and the beginning address of the partition.

VIRT

If VIRT is specified, the sort/merge program will not attempt to fix pages when running in virtual mode. It may be necessary to specify VIRT to prevent interference with other jobs running simultaneously, or to allow a user-written routine to fix pages. VIRT should be avoided wherever possible since it has an unfavorable effect on sort/merge performance. VIRT is ignored when the sort/merge program is running in real mode. The value in SORT-CORE-SIZE will be ignored if the OPTION clause is specified.

ALTWK Option

ALTWK specifies an alternate work drive (tape only) in a sorting job. This doubles the maximum input file size allowed. The address of the alternate device must be different from the address of all other devices used in the job.

ERASE Option

ERASE specifies that work data sets used during a sorting operation are to be erased at the end of the job. It is ignored if 2400-series tapes are used for work areas. If the sort operation terminates abnormally,

- ERASE will be performed unless the checkpoint facility has been specified;
- if ERASE is performed, and if a workfile has been pooled with output, the output file will also be erased.

Note that the sort program does not close work data sets, even when terminating normally.

ROUTE Option

$$\begin{bmatrix}\texttt{,ROUTE=LST}\\ \texttt{,ROUTE=LOG}\end{bmatrix}$$

,ROUTE=LST specifies that messages are to be routed to the SYSLST file by the sort/merge program. Messages requiring operator intervention will also be printed on SYSLOG if allocated to a DOS/VS supported console device.

,ROUTE=LOG specified that messages are to be routed to the console.

Note: The default is assumed until the OPTION card has been read.

SORTWK Option

$$\begin{bmatrix}\texttt{,SORTWK=work}\\ \texttt{,SORTWK=(work}_1\texttt{,...work)}\end{bmatrix}$$

This operand specifies the logical unit numbers associated with the work files. The parameters within parentheses must be replaced by symbolic unit numbers of a maximum of three significant digits from 1 to 221, or by a comma. When a comma is coded, or if the operand is omitted, the sort program will use the default assignment.

SORT-OPTION Clause Examples

SORT-OPTION is SRTOPTN where SRTOPTN is defined in working-storage section. At entry to SORT/MERGE, the contents of SRTOPTN is as follows.

Example 1:

```
OPTION PRINT=ALL,STORAGE=26384,LABEL=(,,U)
```

All messages are requested, the virtual storage available to the sort/merge program is 26,384 bytes, and the work volume is unlabeled.

Example 2:

```
OPTION STORAGE=32K,ERASE,ROUTE=LST,
   SORTWK=(005,006)
```

The PRINT option is not specified, so all messages will be printed by default. The storage available to the sort/merge program is 32K bytes. Standard labels, by default, are assumed for all files. The data sets used by sort are to be erased on completion of the sort operation. All messages are to be routed to the printer. The logical numbers of the work files are SYS005 and SYS006.

Example 3:

```
OPTION SORTWK=(0 10,11,12,,14,15),ALTWK
```

1. Assume work=3 (specified on the SELECT statement associated with the SD file); then, using M from Figure 58.2 (M=3) since, in this example, no overide for the alternate work unit is specified, allocate as follows:

   SYS010, SYS011 and SYS012 are the logical unit numbers of the work files. SYS004 is the logical unit for the alternate work device by default. SYS(M+1)=SYS(3+1)=SYS004.

   SYS014 and SYS015 are not used in this application since WORK=3.

   This example shows how the values interact. The example may be understood as showing a sort operation which was set up to run with five work files, but which for this particular run, has only three work files. (Note the assumption that work=3.)

2. Assume WORK=5, then

   SYS010, SYS011, SYS012, SYS004, and SYS014 are the logical unit numbers of the work files. SYS015 is the logical unit for the alternate device.

Example 4:

A convenient way to specify the OPTION card at execution time is to use the card as a data card on SYSIPT and in the program specify

```
SORT-OPTION IS EXEC-SORT
     .
     .
     .
ACCEPT EXEC-SORT FROM SYSIPT.
```

OUTPUT FILE STATEMENTS

The TLBL or DLBL statement file-name must be SORTOUT. Multivolume and/or multiextent output on disk is accomplished by using DOS/VS standards: one DLBL card is supplied for the entire file followed by one EXTENT card for each separate extent that the file occupies on the disk pack or packs. Where the output file is a direct-access multiextent file, only the first EXTENT statement need contain the specified or defaulted symbolic unit name for the output file. Other EXTENT statements may specify any valid symbolic unit name. Figure 58.2 gives the file-names and default symbolic unit names in the sort/merge program.

Use of Device	Filename	Symbolic Unit Name
Work	SORTWK1 . . . SORTWK9	SYS001 . . . SYS(M)
ALTWK	SORTALT	SYS(M+1)
M=the number of work files, as specified in the SELECT statement for the SD file.		

Figure 58.2. File Name and Default Symbolic Unit Names

Statement	Operands	Comments
OPTION	PRINT={ALL|NONE|CRITICAL} or PRINT	Default=ALL
	STORAGE=n|(n,VIRT)|(nK,....)	Default. See discussion.
	LABEL=(,,work)	Default=standard labels
	ALTWK	
	ERASE	
	ROUTE={LST|LOG}	Default Ph0 msg on printer and console and Ph1-3 on console.
	SORTWK=$\left\{\begin{array}{l}\text{work}_1\\ (\text{work}_1,\text{....work })\end{array}\right\}$	Default=(1,2,...m)

Figure 58.3. SUMMARY OF SORT-OPTION Operands

SORT DIAGNOSTIC MESSAGES

The messages generated by the Sort/Merge Feature are listed in the sort publications referenced in the preface.

LINKAGE WITH THE SORT/MERGE FEATURE

To initiate a sort or merge operation, the COBOL object program includes the object time subroutines ILBDSRT0 and ILBDMRG0 and transfers control to them.

If the INPUT PROCEDURE option of the SORT statement is specified in the source program, exit E15 of the Sort/Merge program is used. At this exit, the record released by the programmer is passed to the Sort/Merge program. Since a dummy Input Procedure will be generated by the compiler when the USING option is specified, records in the USING file are also passed to the Sort/Merge program at exit E15. Records in the USING file of a Merge operation are passed at exit E32.

If the OUTPUT PROCEDURE option of the SORT statement is specified, exit E35 of the Sort/Merge program is used. At this exit, the record returned by the Sort/Merge program is passed to the programmer. Since a dummy Output Procedure is generated by the compiler when the GIVING option is specified, records are also returned at exit E35 and written on this file. Exit E32 is used for the output procedure option of the MERGE statement.

Completion Codes

The Sort/Merge program returns a completion code upon termination and this code is stored in the COBOL special register SORT-RETURN. The codes are:

0 --	Successful completion of Sort/Merge
02 --	Invalid OPEN -- USING file
04 --	Permanent I/O error -- USING file
06 --	Invalid OPEN -- GIVING file
08 --	Permanent I/O error -- GIVING file
10 --	Boundary violation -- GIVING file
12 --	Duplicate or out of sequence key -- GIVING file
16 --	Unsuccessful completion of Sort/Merge

Successful Completion: When a Sort/Merge application has been successfully executed, a completion code of zero is returned and the sort operation terminates.

Unsuccessful Completion: If the Sort program encounters an error during execution that will not allow it to complete successfully, it returns a completion code of 16 and terminates. (A possible error is an uncorrectable input/output error.) The sort publications contain a detailed description of the conditions under which this termination will occur.

The user may test the SORT-RETURN register for successful termination of the sort operation, as shown in the following example:

```
SORT SALES-RECORDS ON ASCENDING KEY,
   CUSTOMER-NUMBER, DESCENDING KEY DATE,
   USING FN-1, GIVING FN-2.


IF SORT-RETURN NOT EQUAL TO ZERO, DISPLAY
   "SORT UNSUCCESSFUL" UPON CONSOLE, STOP
   RUN.
```

Cataloging a Sort Program

When the CATAL option is used to catalog a sort program, the following should be observed:

- To avoid duplicate names when selecting a catalog name for his program, the programmer must be aware of the naming convention used by the compiler to generate the name of the dummy phase into which the phases of the Sort/Merge program will subsequently be loaded.

Naming Convention: The compiler generates the phase card for the dummy phase using the following convention:

- If the PROGRAM-ID name is 6, 7, or 8 characters in length, the dummy phase name consists of the first 6 characters plus 2 zero characters.

- If the PROGRAM-ID name is less than 6 characters in length, the name is padded with zeros to 8 characters.

- Since the system expects the first character of PROGRAM-ID to be alphabetic, the first character, if numeric, is converted as follows:

  ```
  0   -> J
  1-9 -> A-I
  ```

  The hyphen is converted to zero if it appears as the second through eighth character.

CHECKPOINT/RESTART DURING A SORT

The Checkpoint/Restart Feature is available to the programmer using the COBOL SORT statement. The programmer uses the RERUN clause to specify that checkpoints should be taken during program execution. The control statement requirements for taking a checkpoint are discussed in the section entitled "Program Checkout." Checkpoint/Restart is not available during a merge operation.

The system-name specified in the RERUN clause as the sort checkpoint device must not be the same as any system-name used in the source language ASSIGN clause, but follows the same rules of formation.

The RERUN' clause is fully described in the publication IBM DOS Full American National Standard COBOL.

USING SORT IN A MULTIPHASE ENVIRONMENT

When the Sort program is invoked in a multiphase environment, the following should be noted:

1. It is the programmer's responsibility to ensure that the COBOL program containing the SORT statement is the highest phase in storage.

2. If two programs are compiled, link edited, and executed together, only one program may use the Sort feature. If both programs require Sort, the programs can be compiled separately and then the decks must be organized so that the dummy phase cards for Sort are both together at the end of the deck before they are link edited and executed.

3. If Debug and Sort are used together, the Debug modules must be included in the root phase.

REPORT Clause in a File Description (FD) Entry

A given report-name may appear in a maximum of two file description entries. The file description entries need not have the same characteristics, but both must be standard sequential. If the same report-name is specified in two file description entries, the report will be written on both files. For example:

```
ENVIRONMENT DIVISION.
    SELECT FILE-1  ASSIGN SYS005-UR-1403-S.
    SELECT FILE-2  ASSIGN SYS001-UT-2400-S.
    .
    .
DATA DIVISION.
FD  FILE-1   RECORDING MODE F
             RECORD CONTAINS 121 CHARACTERS
             REPORT IS REPORT-A.
FD  FILE-2   RECORDING MODE V
             RECORD CONTAINS 101 CHARACTERS
             REPORT IS REPORT-A.
```

For each GENERATE statement, the records for REPORT-A will be written on FILE-1 and FILE-2, respectively. The records on FILE-2 will not contain columns 102 through 121 of the corresponding records on FILE-1.

Summing Techniques

Execution time of an object program can be decreased by keeping in mind that Report Writer source coding is treated as though the programmer had written the program in COBOL without the Report Writer feature. Therefore, a complex source statement or series of statements will generally be executed faster than simple statements that perform the same function. The following example shows two coding techniques for the Report Section of the Data Division. Method 2 uses the more complex statements.

```
RD...CONTROLS ARE YEAR MONTH WEEK DAY.
```

Method 1:

```
01 TYPE CONTROL FOOTING YEAR.
   02 SUM COST.
01 TYPE CONTROL FOOTING MONTH.
   02 SUM COST.
01 TYPE CONTROL FOOTING WEEK.
   02 SUM COST.
01 TYPE CONTROL FOOTING ADAY.
   02 SUM COST.
```

Method 2:

```
01 TYPE CONTROL FOOTING YEAR.
   02 SUM A.
01 TYPE CONTROL FOOTING MONTH.
   02 A SUM B.
01 TYPE CONTROL FOOTING WEEK.
   02 B SUM C.
01 TYPE CONTROL FOOTING ADAY.
   02 C SUM COST.
```

Method 2 will execute faster. One addition will be performed for each day, one more for each week, and one for each month. In Method 1, four additions will be performed for each day.

Use of SUM

Unless each identifier is the name of a SUM counter in a TYPE CONTROL FOOTING report group at an equal or lower position in the control hierarchy, the identifier must be defined in the File, Working-Storage, or Linkage Sections as well as in a TYPE DETAIL report group as a source item or no summing will occur. A SUM counter is algebraically incremented just before presentation of the TYPE DETAIL report group in which the item being summed appears as a source item or the item being summed appeared in a SUM clause that contained an UPON option for this DETAIL report group. This is known as SOURCE-SUM correlation. In the following example, SUBTOTAL is incremented only when DETAIL-1 is generated.

```
FILE SECTION.
       .
       .
       .
    02 NO-PURCHASES PICTURE 99.
       .
       .
       .
REPORT SECTION.
01 DETAIL-1 TYPE DETAIL.
    02 COLUMN 30 PICTURE 99 SOURCE
       NO-PURCHASES.
       .
       .
       .
01 DETAIL-2 TYPE DETAIL.
       .
       .
       .
01 ADAY TYPE CONTROL FOOTING
    LINE PLUS 2.
       .
       .
       .
    02 SUBTOTAL COLUMN 30 PICTURE 999
       SUM NO-PURCHASES.
       .
       .
       .
01 MONTH TYPE CONTROL FOOTING
    LINE PLUS 2 NEXT GROUP
    NEXT PAGE.
```

SUM Routines

A SUM routine is generated by the Report Writer for each DETAIL report group of the report. The operands included for summing are determined as follows:

1. The SUM operand(s) also appears in a SOURCE clause(s) for the DETAIL report group.

2. The UPON detail-name option was specified in the SUM clause. In this case, all the operands are included in the SUM routine for only that DETAIL report group, even if the operand appears in a SOURCE clause in other DETAIL report groups.

When a GENERATE detail-name statement is executed, the SUM routine for that DETAIL report group is executed in its logical sequence. When GENERATE report-name statement is executed and the report contains more than one DETAIL report group, the SUM routine is executed for each one. The SUM routines are executed in the sequence in which the DETAIL report groups are specified.

The following two examples show the SUM routines that are generated by the Report Writer. Example 1 illustrates how operands are selected for inclusion in the routine on the basis of simple SOURCE-SUM correlation. Example 2 illustrates how operands are selected when the UPON detail-name option is specified.

Example 1: The following statements are coded in the Report Section:

```
01 DETAIL-1 TYPE DE ...
   02 ...SOURCE A.
                .
                .
                .
01 DETAIL-2 TYPE DE ...
   02 ...SOURCE B.
   02 ...SOURCE C.
                .
                .
                .
01 DETAIL-3 TYPE DE ...
   02 ...SOURCE B.
                .
                .
                .
01 TYPE CF ...
   02 SUM-CTR-1 ...SUM A, B, C.
                .
                .
                .
01 TYPE CF ...
   02 SUM-CTR-2 ...SUM B.
```

A SUM routine is generated for each DETAIL report group, as follows:

SUM-ROUTINE FOR DETAIL-1

```
   REPORT-SAVE
      ADD A TO SUM-CTR-1.
   REPORT-RETURN
```

SUM-ROUTINE FOR DETAIL-2

```
   REPORT-SAVE
      ADD B TO SUM-CTR-1.
      ADD C TO SUM-CTR-1.
      ADD B TO SUM-CTR-2.
   REPORT-RETURN
```

SUM-ROUTINE FOR DETAIL-3

```
   REPORT-SAVE
      ADD B TO SUM-CTR-1.
      ADD B TO SUM-CTR-2.
   REPORT-RETURN
```

Example 2: This example uses the same coding as Example 1, with one exception: the UPON detail-name option is used for SUM-CTR-1, as follows:

```
01 TYPE CF ...
   02 SUM-CTR-1 ...SUM A, B, C
      UPON DETAIL-2.
```

The following SUM routines would then be generated instead of those shown in the previous example:

SUM Routine for DETAIL-1

```
   REPORT-SAVE
   REPORT-RETURN
```

SUM Routine for DETAIL-2

```
   REPORT-SAVE
      ADD A TO SUM-CTR-1.
      ADD B TO SUM-CTR-1.
      ADD C TO SUM-CTR-1.
      ADD B TO SUM-CTR-2.
   REPORT-RETURN
```

SUM Routine for DETAIL-3

```
   REPORT-SAVE
      ADD B TO SUM-CTR-2.
   REPORT-RETURN
```

Output Line Overlay

The Report Writer output line is created using an internal REDEFINES specification, indexed by integer-1. No check is made to prevent overlay on any line. For example:

```
    02  COLUMN 10  PICTURE X(23)
        VALUE "MONTHLY SUPPLIES REPORT".
    02  COLUMN 12  PICTURE X(9)
        SOURCE CURRENT-MONTH.
```

A length of 27 in column 10, followed by a specification for column 12, will cause field overlay when this line is printed.

Page Breaks

The Report Writer page break routine operates independently of the routines that are executed after any control breaks (except that a page break will occur as the result of a LINE NEXT PAGE clause). Thus, the programmer should be aware of the following facts:

1. A Control Heading is not printed after a Page Heading except for first generation. If the programmer wishes to have the equivalent of a Control Heading at the top of each page, he must include the information and data to be printed as part of the Page Heading. Since only one Page Heading may be specified for each report, he should be selective in considering his Control Heading because it will be the same for each page, and may be printed at inappropriate times (see "Control Footings and Page Format" in this chapter).

2. GROUP INDICATE items are printed after page and control breaks. Figure 56 contains a GROUP INDICATE clause and illustrates the execution output.

```
REPORT SECTION.
   .
   .
   .
01  DETAIL-LINE TYPE IS DETAIL LINE
   NUMBER IS PLUS 1.
   02  COLUMN IS 2 GROUP INDICATE
       PICTURE IS A(9) SOURCE IS
       MONTHNAME OF RECORD-AREA (MONTH).
   .
   .
   .

(Execution Output)
-------------------------------------------
JANUARY    15   A00...
                A02...

PURCHASES AND COST...
-------------------------------------------
JANUARY    21   A03...
                A03...
```

Figure 59. Sample of GROUP INDICATE Clause and Resultant Execution Output

WITH CODE Clause

When more than one report is being written on a file and the reports are to be selectively written, a unique 1-character code must be given for each report. A mnemonic-name is specified in the RD-level entry for each report and is associated with the code in the Special-Names paragraph of the Environment Division.

Note: If a report is written with the CODE option, the report should not be written directly on a printer device.

This code will be written as the first character of each record that is written on the file. When the programmer wishes to write a report from this file, he needs only to read a record, check the first character for the desired code, and have it printed if the desired code is found. The record should be printed starting from the third character, as illustrated in Figure 60.

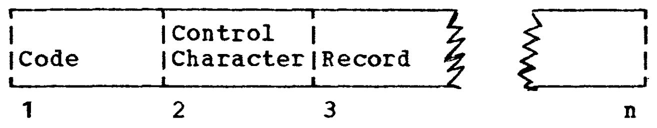


Figure 60. Format of a Report Record When the CODE Clause is Specified

The following example shows how to create and print a report with a code of A. A Report Writer program contains the following statements:

```
ENVIRONMENT DIVISION.
.
.
SPECIAL-NAMES.  "A" IS CODE-CHR-A
                "B" IS CODE-CHR-B.
.
.
DATA DIVISION.
.
.
REPORT SECTION.
RD  REP-FILE-A   CODE CODE-CHR-A ...
.
.
RD  REP-FILE-B   CODE CODE-CHR-B ...
```

A second program could then be used to print only the report with the code of A, as follows:

```
DATA DIVISION.
FD  RPT-IN-FILE
    RECORD CONTAINS 122 CHARACTERS
    LABEL RECORDS ARE STANDARD
    DATA RECORD IS RPT-RCD.
01  RPT-RCD.
    05  CODE-CHR        PICTURE X.
    05  PRINT-PART.
        10  CTL-CHR     PICTURE X.
        10  RECORD-PART PICTURE X(120).
FD  PRINT-FILE
    RECORD CONTAINS 121 CHARACTERS
    LABEL RECORDS ARE STANDARD
    DATA RECORD IS PRINT-REC.
01  PRINT-REC.
    05  FILLER            PICTURE X(121).
.
.
PROCEDURE DIVISION.
.
.
LOOP.  READ RPT-IN-FILE AT END
           GO TO CONTINUE.
           IF CODE-CHR = "A"
           WRITE PRINT-REC FROM PRINT-PART
           AFTER POSITIONING CTL-CHR LINES.
           GO TO LOOP.
CONTINUE.
           .
           .
           .
```

Control Footings and Page Format

Depending on the number and size of Control Footings (as well as the page depth of the report), all of the specified Control Footings may not be printed on the same page if a control break occurs for a high-level control. When a page condition is detected before all required Control Footings are printed, the Report Writer will print the Page Footing (if specified), skip to the next page, print the Page Heading (if specified) and then continue to print Control Footings.

If the programmer wishes all of his Control Footings to be printed on the same page, he must format his page in the RD-level entry for the report (by setting the LAST DETAIL integer to a sufficiently low line number) to allow for the necessary space.

NEXT GROUP Clause

Each time a CONTROL FOOTING report group with a NEXT GROUP clause is printed, the clause is activated only if the report group is associated with the control that causes the break. This is illustrated in Figure 61.

```
RD  EXPENSE-REPORT CONTROLS ARE FINAL,
    MONTH, ADAY
     .
     .
     .
01  TYPE CONTROL FOOTING DAY
    LINE PLUS 1 NEXT GROUP
    NEXT PAGE.
     .
     .
     .
01  TYPE CONTROL FOOTING MONTH
    LINE PLUS 1 NEXT GROUP
    NEXT PAGE.
     .
     .
     .

(Execution Output)

EXPENSE REPORT
     .
     .
     .
January 31.........29.30
    (Output for CF ADAY)
January total.....131.40
    (Output for CF MONTH)
```

Figure 61. Activating the NEXT GROUP Clause

Note: The NEXT GROUP NEXT PAGE clause for the Control Footing DAY is not activated.

Floating First Detail

The first presentation of a body group (PH, PF, CH, CF, DE) that contains a relative line as its first line will have its relative line spacing suppressed; the first line will be printed on either the value of FIRST DETAIL or INTEGER PLUS 1 of a NEXT GROUP clause from the preceding page. For example:

1. If the following body group was the last to be printed on a page

   `01  TYPE CF NEXT GROUP NEXT PAGE`

   then this next body group

   `01  TYPE DE LINE PLUS 5`

   would be printed on value of FIRST DETAIL (in PAGE clause).

2. If the following body group was the last to be printed on a page

   `01  TYPE CF NEXT GROUP LINE 12`

   and after printing, line-counter = 40, then this next body group

   `01  TYPE DETAIL LINE PLUS 5`

   would be printed on line 12 + 1 (i.e., line 13).

Report Writer Routines

At the end of the analysis of a report description (RD) entry, the Report Writer routines are generated, based on the contents of the RD. Each routine references the compiler-generated card number of its respective RD.

Subscripts

If a subscript is represented by a constant and if the subscripted item is of fixed length, the location of the subscripted data item within the table or list is resolved during compilation.

If a subscript is represented by a data-name, the location is resolved at execution time. The most efficient format in this case is COMPUTATIONAL, with a PICTURE size less than five integers.

The value contained in a subscript is an integer which represents an occurrence number within a table. Every time a subscripted data-name is referenced in a program, the compiler generates up to 16 instructions to calculate the correct displacement. Therefore, if a subscripted data-name is to be processed extensively, move the subscripted item to an unsubscripted work area, do all necessary processing, and then move the item back into the table. Even when subscripts are described as COMPUTATIONAL, subscripting takes time and storage.

Index-names

Index-names are compiler-generated items, one fullword in length, assigned storage in the TGT (Task Global Table). An index-name is defined by the INDEXED BY clause. The value in an index-name represents an actual displacement from the beginning of the table that corresponds to an occurrence number in the table. Address calculation for a direct index requires a maximum of four instructions; address calculation for a relative index requires a few more. Therefore, the use of index-names in referencing tables is more efficient than the use of subscripts. The use of direct indexes is faster than the use of relative indexes.

Index-names can only be referenced in the PERFORM, SEARCH, and SET statements.

Index Data Items

Index data items are compiler-generated storage positions, one fullword in length, that are assigned storage within the COBOL program area. An index data item is defined by the USAGE IS INDEX clause. The programmer can use index data items to save values of index-names for later reference.

Great care must be taken when setting values of index data items. Since an index data item is not part of any table, the compiler is unable to change any displacement value contained in an index-name when an index data item is set to the value of an index-name or another index data item. See the SET statement examples later in this chapter.

Index data items can only be referenced in SEARCH and SET statements.

OCCURS Clause

If indexing is to be used to reference a table element and the Format 2 (SEARCH ALL) statement is also used, the KEY option _must_ be specified in the OCCURS clause. A _table element_ is represented by the subject of an OCCURS clause, and is equivalent to one level of a table. The table element must then be ordered upon the key(s) and data-name(s) specified.

TB
HD

DEPENDING ON Option

If a data item described by an OCCURS clause with the DEPENDING ON _data-name_ option is followed by nonsubordinate data items, a change in the value of _data-name_ during the course of program execution will have the following effects:

1. The size of any group described by or containing the related OCCURS clause will reflect the new value of _data-name_.

2. Whenever a MOVE to a field containing an OCCURS clause with the DEPENDING ON option is executed, the MOVE is done on the basis of the current contents of the object of the DEPENDING ON option.

3. The location of any nonsubordinate items following the item described with the OCCURS clause will be affected by the new value of

data-name. If the programmer wishes to preserve the contents of these items, the following procedure can be used: prior to the change in data-name, move all nonsubordinate items following the variable item to a work area; after the change in data-name, move all the items back.

Note: The value of data-name may change because a move is made to it or to the group in which it is contained; or the value of data-name may change because the group in which it is contained is a record area that has been changed by execution of a READ statement.

For example, assume that the Data Division of a program contains the following coding:

```
     01  ANYRECORD.
         05  A PICTURE S999 COMPUTATIONAL-3.
         05  TABLEA PICTURE S999 OCCURS 100
             TIMES DEPENDING ON A.
         05  GROUPB.

             Subordinate data items.
                     End of record.
```

GROUPB items are not subordinate to TABLEA, which is described by the OCCURS clause. Assuming that WORKB is a work area with the same data structure as GROUPB, the following procedural coding could be used:

```
MOVE GROUPB TO WORKB

Calculate a new value of A

MOVE WORKB TO GROUPB
```

The preceding statements can be avoided by placing the OCCURS clause with the DEPENDING ON option at the end of the record.

Note: data-name can also change because of a change in the value of an item that redefines or renames it. In this case, the group size and the location of nonsubordinate items as described in the two preceding paragraphs cannot be determined.

OCCURS CLAUSE WITH THE DEPENDING ON OPTION

If a record description contains an OCCURS clause with the DEPENDING ON option, the record length is variable. This is true for records described in an FD as well as in the Working-Storage section. A previous chapter discussed four different record formats of non-VSAM files. Three of them, V-mode, U-mode, and S-mode, as well as VSAM files, may contain one or more OCCURS clauses with the DEPENDING ON option.

This section discusses some factors that affect the manipulation of records containing OCCURS clauses with the DEPENDING ON option. The text indicates whether the factors apply to the File or Working-Storage sections, or both.

The compiler calculates the length of V-mode records containing the OCCURS clause with the DEPENDING ON option at three different times, as follows (the first and third applies to FD entries only; the second to both FD and Working-Storage entries):

1. When a file is read and the object of the DEPENDING ON option is within the record.

2. When the object of the DEPENDING ON option is changed as a result of a move to it or any item within its group. (The length is not calculated when a move is made to an item which redefines or renames it.)

   For instance before a group item with an OCCURS DEPENDING ON clause in it can be moved from an I/O area to working storage, the object of the DEPENDING ON clause must be moved separately from the I/O area to the corresponding area in working storage to force initial calculation of the receiving field's length.

   If the object of the DEPENDING ON option is changed outside of the COBOL program, to insure correct length, a dummy move of the DEPENDING ON object must be made upon return to the COBOL program.

3. For an output file, after the record is written, the length is set to maximum to enable a full move of the next record to the buffer. Immediately after the move, the correct length is recalculated as in item 2.

Consider the following example:

```
WORKING-STORAGE SECTION.

77  CONTROL-1     PIC 99.
77  WORKAREA-1    PIC 9(6)V99.
         .
         .
         .
01  SALARY-HISTORY.
    05  SALARY OCCURS 0 TO 10 TIMES
        DEPENDING ON
        CONTROL-1  PIC 9(6)V99.
```

The Procedure Division statement MOVE 5 TO CONTROL-1 will cause a recalculation of the length of SALARY-HISTORY. MOVE SALARY (5) TO WORKAREA-1 will not cause the length to be recalculated.

The compiler permits the occurrence of more than one level-01 record, containing the OCCURS clause with the DEPENDING ON option, in the same FD entry (see Figure 62). For non-VSAM files, if the BLOCK CONTAINS clause is omitted, the buffer size is calculated from the longest level-01 record description entry. In Figure 62, the buffer size is determined by the description of RECORD-1 (RECORD-1 need not be the first record description under the FD).

During the execution of a READ statement, the length of each level-01 record description entry in the FD will be calculated (see Figure 62). The length of the variable portion of each record will be the product of the numeric value contained in the object of the DEPENDING ON option and the length of the subject of the OCCURS clause. In Figure 62, the length of FIELD-1 is calculated by multiplying the contents of CONTROL-1 by the length of FIELD-1; the length of FIELD-2, by the product of the contents of CONTROL-2 and the length of FIELD-2; the length of FIELD-3 by the contents of CONTROL-3 and the length of FIELD-3.

Since the execution of a READ statement makes available only one record type (i.e., RECORD-1 type, RECORD-2 type, or RECORD-3 type), two of the three record descriptions in Figure 62 will be inappropriate. In such cases, if the contents of the object of the DEPENDING ON option does not conform to its picture, the length of the corresponding record will be unpredictable. For the contents of an item to conform to its picture:

- An item described as USAGE DISPLAY must contain external decimal data.
- An item described as USAGE COMPUTATIONAL-3 must contain internal decimal data.
- An item described as USAGE COMPUTATIONAL must contain binary data.
- An item described as signed must contain signed data.
- An item described as unsigned must contain unsigned data.

The following example illustrates the length calculations made by the system when a READ statement is executed:

```
FD
    .
    .
    .
01  RECORD-1.
    05  A  PIC  99.
    05  B  PIC  99.
    05  C  PIC  99  OCCURS 5 TIMES
        DEPENDING ON A.

01  RECORD-2.
    05  D  PIC  XX.
    05  E  PIC  99.
    05  F  PIC  99.
    05  G  PIC  99  OCCURS 5 TIMES
        DEPENDING ON F.

WORKING-STORAGE SECTION.
    .
    .
    .
01  TABLE-3.
    05  H PIC99 OCCURS 10 TIMES DEPENDING
        ON B.

01  TABLE-4.
    05  I PIC99 OCCURS 10 TIMES DEPENDING
        ON E.
```

When a record is read, lengths are determined as follows:

1. The length of C is calculated using the contents of field A. The length of RECORD-1=A+B+C.
2. The length of G is calculated using the contents of field F. The length of RECORD-2=D+E+F+G.
3. The length of TABLE-3 is calculated using the contents of field B.
4. The length of TABLE-4 is calculated using the contents of field E.

The programmer should be aware of several characteristics of the previously cited length calculations. The following example illustrates a group item (i.e., REC-1) whose subordinate items contain an OCCURS clause with the DEPENDING ON option and the object of that DEPENDING ON option.

```
WORKING-STORAGE SECTION.
01  REC-1.
    05  FIELD-1 PIC 9.
    05  FIELD-2 OCCURS 5 TIMES DEPENDING ON
                FIELD-1  PIC X(5).
```

```
FD  INPUT-FILE
    .
    .
    DATA RECORDS ARE RECORD-1 RECORD-2 RECORD-3.

01  RECORD-1.
    05  CONTROL-1              PIC 99.
    05  FIELD-1 OCCURS 0 TO 10 TIMES DEPENDING ON CONTROL-1  PIC 9(5).

01  RECORD-2.
    05  CONTROL-2              PIC 99.
    05  FIELD-2 OCCURS 1 TO 5 TIMES DEPENDING ON CONTROL-2  PIC 9(4).

01  RECORD-3.
    05  FILLER                 PIC XX.
    05  CONTROL-3              PIC 99.
    05  FIELD-3 OCCURS 0 TO 10 TIMES DEPENDING ON CONTROL-3  PIC X(4).
```

Figure 62. Calculating Record Lengths When Using the OCCURS Clause with the DEPENDING ON Option

```
01  REC-2.
    05  REC-2-DATA  PIC X(50).
```

The results of executing a MOVE to the group item REC-1 will be affected by the following:

- The length of REC-1 may have been calculated at some time prior to the execution of this MOVE statement. The user should make sure that REC-1 reflects the correct length.

- The length of REC-1 may never have been calculated at all, and the result of the MOVE will be unpredictable.

- After the move, since the contents of FIELD-1 have been changed, an attempt will be made to recalculate the length of REC-1. Correct recalculation, however, will only be made if the new contents of FIELD-1 conform to its picture (i.e., USAGE DISPLAY must contain an external decimal item, USAGE COMPUTATIONAL-3 must contain an internal decimal item and USAGE COMPUTATIONAL must contain a binary item. An item described as signed must contain signed data, and an item described as unsigned must contain unsigned data). In the preceding example, if FIELD-1 does not contain an external decimal item, the length of REC-1 will be unpredictable.

Note: According to the COBOL description, FIELD-2 can occur a maximum of five times. If, however, FIELD-1 contains an external decimal item whose value exceeds five, the length of REC-1 will still be calculated. One possible consequence of this invalid calculation will be encountered if the programmer attempts to initialize REC-1 by moving zeros or spaces to it. This initialization would inadvertently delete part of the adjacent data stored in REC-2.

The following discussion applies to updating a record containing an OCCURS clause with the DEPENDING ON option and at least one other subsequent entry. In this case, the subsequent entry is another item containing an OCCURS clause with the DEPENDING ON option.

```
WORKING-STORAGE SECTION.
01  VARIABLE-REC.
    05  FIELD-A      PIC X(10).
    05  CONTROL-1    PIC 99.
    05  CONTROL-2    PIC 99.
    05  VARY-FIELD-1 OCCURS 10 TIMES
        DEPENDING ON CONTROL-1 PIC X(5).
    05  TEMP.
        06  VARY-FIELD-2 OCCURS
            10 TIMES DEPENDING ON
            CONTROL-2  PIC X(9).
01  STORE-VARY-FIELD-2.
    05  VARY-FLD-2 OCCURS 10 TIMES
        DEPENDING ON CONTROL-2 PIC X(9).
```

Assume that CONTROL-1 contains the value 5 and VARY-FIELD-1 contains 5 entries.

In order to add a sixth field to VARY-FIELD-1 the following steps are required:

```
MOVE TEMP TO STORE-VARY-FIELD-2.
ADD 1 TO CONTROL-1.
MOVE 'additional field' TO VARY-FIELD-1
  (CONTROL-1).
MOVE STORE-VARY-FIELD-2 TO TEMP.
```

SET Statement

The SET statement is used to assign values to index-names and to index data items.

When an index-name is set to the value of a literal, identifier, or an index-name from another table element, it is set to an actual displacement from the beginning of the table that corresponds to the occurrence number indicated by the second operand in the statement. The compiler performs the necessary calculations. If an index-name is set to another index-name for the same table, the compiler need make no conversion of the actual displacement value contained in the second operand.

However, when an index data item is set to another index data item or to an index-name, or when an index-name is set to an index data item, the compiler is unable to change any displacement value it finds, since an index data item is not part of any table. Thus, no conversion of values can take place. Remember this to avoid making programming errors.

For example, suppose that a table has been defined as:

```
01 A.
   05 B OCCURS 2 INDEXED BY I1, I5.
      10 C OCCURS 2 INDEXED BY I2, I6.
         15 D OCCURS 3 INDEXED BY I3, I4.
            20 E PIC X(20).
            20 F PIC 9(5).
```

The table appears in storage as shown in Figure 63.

Suppose that a reference to D (2, 2, 3) is necessary. The following method is incorrect:

```
SET I3 TO 2.
SET INDX-DATA-ITM TO I3.
SET I3 UP BY 1.
SET I2, I1 TO INDX-DATA-ITM.
MOVE D (I1, I2, I3) TO WORKAREA.
```

The value contained in I3 after the first SET statement is 25, which represents the beginning point of the second occurrence of D. When the second SET statement is executed, the value 25 is placed in INDX-DATA-ITM, and the fourth SET statement moves the value 25 into I2 and I1. The third SET statement increases the value in I3 to 50. The calculation for the address D (I1, I2, I3) would then be as follows:

```
(address of D (1, 1, 1)) + 25 + 25 + 50
  = (address of D (1, 1, 1)) + 100
```

This is not the address of D (2, 2, 3).

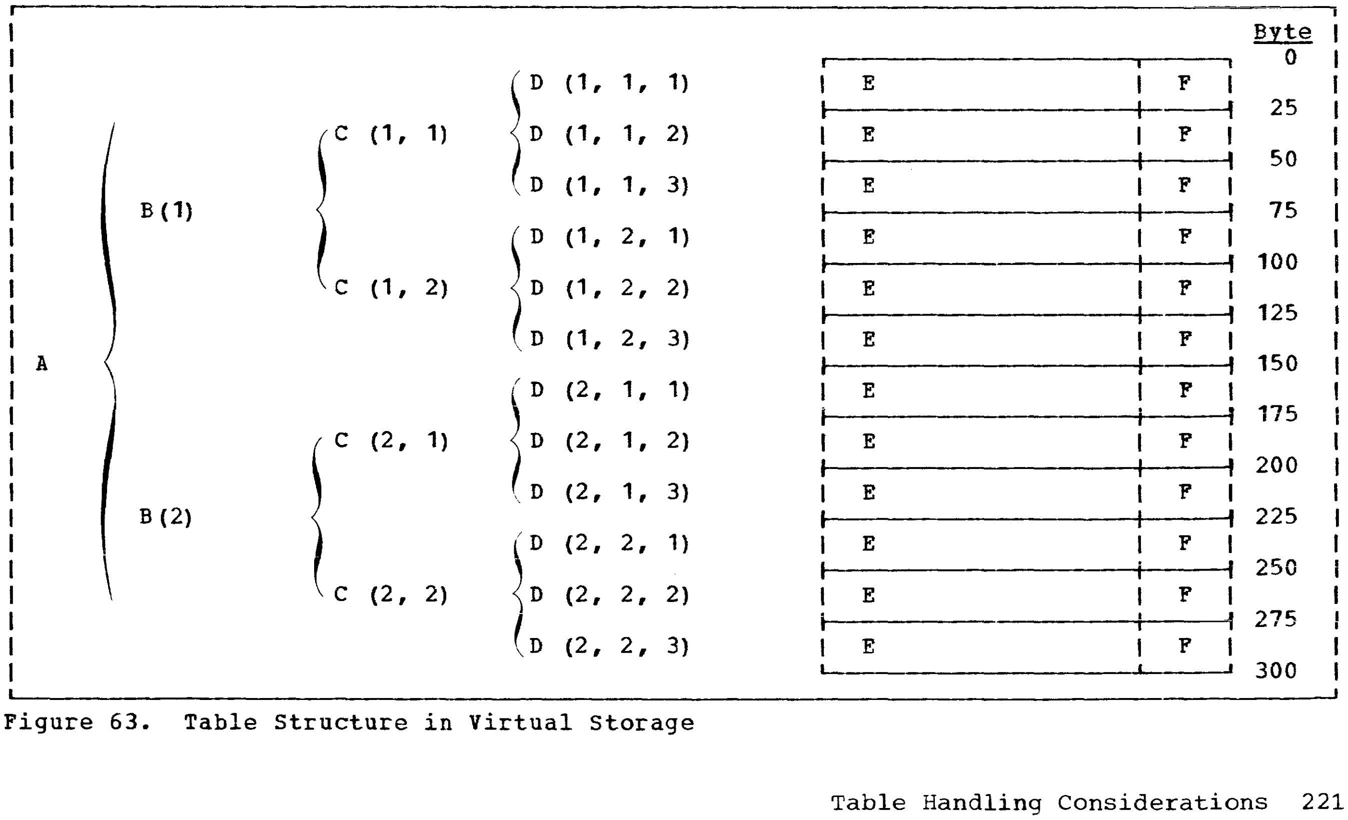


Figure 63. Table Structure in Virtual Storage

The following method will find the correct address:

```
SET I3 TO 2.
SET I2, I1 TO I3.
SET I3 UP BY 1.
```

In this case, the first SET statement places the value 25 in I3. Since the compiler is able to calculate the lengths of B and C, the second SET statement places the value 75 in I2, and the value 150 in I1. The third SET statement places the value 50 in I3. The correct address calculation will be:

```
(address of D (1, 1, 1)) + 150 + 75 + 50
  = (address of D (1, 1, 1)) + 275
```

The rules for the SET statement are shown in Table 33.

Use care when setting the value of index-names associated with tables described as OCCURS DEPENDING ON. If the table entry length is changed, the value contained within the index-name will become invalid unless a new SET statement corrects it.

Table 33. Rules for the SET Statement

Receiving \ Sending	Index-name	Index data item	Identifier or Literal
Index-name	Set to value corresponding to occurrence number[1]	Move without conversion	Set to value corresponding to occurrence number
Index data item	Move without conversion	Move without conversion	Illegal
Identifier	Set to occurrence number represented by index-name	Illegal	Illegal

[1]If index-names refer to the same table element, move without conversion.

TBL
HDL

SEARCH Statement

Only one level of a table (a table element) can be referenced with one SEARCH statement. Note that SEARCH statements cannot be nested, since an imperative-statement must follow the WHEN condition, and the SEARCH statement is itself conditional.

To write a series of statements that will search the 3-dimensional table defined in the discussion of the SET statement, the programmer could write:

```
.
.
.

77   COMPARAND1 PIC X(5).
77   COMPARAND2 PIC 9(5).

01   A.
   02 B OCCURS 2 INDEXED BY I1 I5.
      03 C OCCURS 2 INDEXED BY I2 I6.
         04 D OCCURS 3 INDEXED BY I3 I4.
            05 E PIC X(5).
            05 F PIC 9(5).
.
.
.

(Initialize COMPARAND1 and COMPARAND2)

   PERFORM SEARCH-TEST1 THRU SEARCH-EXIT1
   VARYING I1 FROM 1 BY 1 UNTIL I2 IS
   GREATER THAN 2.
ENTRY-NOENTRY1.
   GO TO ERROR-RECOVERY1.

SEARCH-TEST1.
   SET I3 TO 1.
   SEARCH D WHEN E (I1, I2, I3) =
      COMPARAND1 AND F (I1, I2, I3) =
      COMPARAND2
   SET I5 TO I1
   SET I6 TO I2
   SET I2 TO 3
   SET I1 TO 3
   ALTER ENTRY-NOENTRY1 TO PROCEED
      TO ENTRY-PROCESSING1.
SEARCH-EXIT1. EXIT.
.
.
.
ERROR-RECOVERY1.
.
.
.
ENTRY-PROCESSING1.
   MOVE E (I5, I6, I3) TO OUTAREA1.
   MOVE F (I5, I6, I3) TO OUTAREA2.
.
.
.
```

The PERFORM statement varies the indexes (I1 and I2) associated with table elements B and C; the SEARCH statement varies index I3 associated with table element D.

The values of I1 and I2 that satisfy the WHEN conditions of the SEARCH statement are saved in I5 and I6. I1 and I2 are then both set to 3, so that upon return from the SEARCH statement, control will fall through the PERFORM statement to the GO TO statement.

Subsequent references to the desired occurrence of table elements E and F make use of the index-names I5 and I6 in which the correct value was saved.

Since a SEARCH verb results in the examination of the individual elements in the named table, the XREF or SXREF for a SEARCH will reference the element name for the table rather than the table itself. LISTER could provide the source cross-reference material that might be desired.

Format 1 SEARCH statements perform a serial search of a table. If it is certain that the "found" condition is beyond some intermediate point in the table, the index-names can be set at that point and only that part of the table be searched; this speeds up execution. If the table is large and must be searched from the first occurrence to the last, Format 2 (SEARCH ALL) is more efficient than Format 1, since it uses a binary search technique; however, the table must then be ordered.

In Format 1, the VARYING option allows the programmer to:

- Vary an index-name other than the first index-name stated for this table element. Thus, with two SEARCH statements, each using a different index-name, more than one value can be referenced in the same table element for comparisons, etc.

- Vary an index-name from another table element. In this case, the first index-name specified for this table is used for the SEARCH, and the index-name specified in the VARYING option is incremented at the same time. Thus, the programmer can search two table elements at once.

SEARCH ALL Statement

The SEARCH ALL statement is used to search an entire table for an item without having to write a loop procedure. For example, a programmer-defined table may be the following:

```
01  TABLE.
    05  ENTRY-IN-TABLE OCCURS 90 TIMES
        ASCENDING KEY-1,KEY-2
        DESCENDING KEY-3
        INDEXED BY INDEX-1.
        10  PART-1  PICTURE 9(2).
        10  KEY-1   PICTURE 9(5).
        10  PART-2  PICTURE 9(6).
        10  KEY-2   PICTURE 9(4).
        10  PART-3  PICTURE 9(33).
        10  KEY-3   PICTURE 9(5).
```

A search of the entire table can be initiated with the following instruction:

```
SEARCH ALL ENTRY-IN-TABLE AT END GO TO
NOENTRY WHEN KEY-1 (INDEX-1) = VALUE-1
AND KEY-2 (INDEX-1) = VALUE-2 AND KEY-3
(INDEX-1) = VALUE-3 MOVE PART-1
(INDEX-1) TO OUTPUT-AREA.
```

The preceding instructions will execute a search on the given array TABLE, which contains 90 elements of 55 bytes and 3 keys. The primary and secondary keys (KEY-1 and KEY-2) are in ascending order whereas the least significant key (KEY-3) is in descending order. If an entry is found in which the three keys are equal to the given values (i.e., VALUE-1, VALUE-2, VALUE-3), PART-1 of that entry will be moved to OUTPUT-AREA. If matching keys are not found in any of the entries in TABLE, the NOENTRY routine is entered.

If a match is found between a table entry and the given values, the index (INDEX-1) is set to a value corresponding to the relative position within the table of the matching entry. If no match is found, the index remains at the setting it had when execution of the SEARCH ALL statement began.

Note: It is more efficient to test keys in order of significance (i.e., KEY-1 should be specified before KEY-2 in the WHEN statement). The WHEN statement can only test for equality, and only one side of the equation may be a key.

In Format 2, the SEARCH ALL statement, the table must be ordered on the key(s) specified in the OCCURS clause. Any key may be specified in the WHEN condition, but all preceding data-names in the KEY option

In Format 1, the WHEN condition can be any relation condition and there can be more than one. If multiple WHEN conditions are stated, the implied logical connective is OR -- that is, if <u>any</u> one of the WHEN conditions is satisfied, the imperative-statement following the WHEN condition is executed. If <u>all</u> conditions are to be satisfied before exiting from the SEARCH, the compound WHEN condition with AND as the logical connective must be written.

SEARCH ALL Statement

The SEARCH ALL statement is used to search an entire table for an item without having to write a loop procedure. For example, a programmer-defined table may be the following:

```
 01  TABLE.
     05  ENTRY-IN-TABLE OCCURS 90 TIMES
        ASCENDING KEY-1,KEY-2
        DESCENDING KEY-3
        INDEXED BY INDEX-1.
        10  PART-1  PICTURE 9(2).
        10  KEY-1   PICTURE 9(5).
        10  PART-2  PICTURE 9(6).
        10  KEY-2   PICTURE 9(4).
        10  PART-3  PICTURE 9(33).
        10  KEY-3   PICTURE 9(5).
```

A search of the entire table can be initiated with the following instruction:

```
SEARCH ALL ENTRY-IN-TABLE AT END GO TO
NOENTRY WHEN KEY-1 (INDEX-1) = VALUE-1
AND KEY-2 (INDEX-1) = VALUE-2 AND KEY-3
(INDEX-1) = VALUE-3 MOVE PART-1
(INDEX-1) TO OUTPUT-AREA.
```

The preceding instructions will execute a search on the given array TABLE, which contains 90 elements of 55 bytes and 3 keys. The primary and secondary keys (KEY-1 and KEY-2) are in ascending order whereas the least significant key (KEY-3) is in descending order. If an entry is found in which the three keys are equal to the given values (i.e., VALUE-1, VALUE-2, VALUE-3), PART-1 of that entry will be moved to OUTPUT-AREA. If matching keys are not found in any of the entries in TABLE, the NOENTRY routine is entered.

If a match is found between a table entry and the given values, the index (INDEX-1) is set to a value corresponding to the relative position within the table of the matching entry. If no match is found, the index remains at the setting it had when execution of the SEARCH ALL statement began.

<u>Note</u>: It is more efficient to test keys in order of significance (i.e., KEY-1 should be specified before KEY-2 in the WHEN statement). The WHEN statement can only test for equality, and only one side of the equation may be a key.

In <u>Format 2</u>, the SEARCH ALL statement, the table must be ordered on the key(s) specified in the OCCURS clause. Any key may be specified in the WHEN condition, but all preceding data-names in the KEY option must also be tested. The test must be an "equal to" (=) condition, and the KEY data-name must be either the subject or object of the condition, or the name of a conditional variable with which the tested condition-name is associated. The WHEN condition can also be a compound condition, formed from one of the simple conditions listed above, with AND as the <u>only</u> logical connective. The KEY data item and the item with which it is compared must be compatible, as given in the rules of the relation test.

Compilation is faster if keys are tested in the SEARCH statement in the same order as they appear in the KEY option.

Note that if KEY entries within the table do not contain valid values, then the results of the binary search will be unpredictable.

Building Tables

When reading in data to build an internal table:

1. Check to make sure the data does not exceed the space allocated for the table.
2. If the data must be in sequence, check the sequence.
3. If the data contains the subscript that determines its position in the table, check the subscript for a valid range.
4. If a fixed-length table is defined larger (for example, 150 entries) than the actual data supplied (for example, 100 data entries), then the table must be initialized to high value for ascending search or low value for descending search.

When testing for the end of a table, use a named value giving the item count, rather than using a literal. Then, if the table must be expanded, only one value need be changed, instead of all references to a literal.

PART IV

This chapter describes the lister feature, a major new facility for optionally producing reformatted source listings with expanded, embedded cross referencing information to increase intelligibility and conserve space. Topics discussed in this chapter include:

- Overall operation of the lister feature
- The output source listing
- The output summary listing
- The optional reformatted output deck
- Using the lister feature

Features of the new source listing include:

- Standard indentation for all Data Division level numbers to show group structure, and for all IF statements and the like in the Procedure Division to show program logic.
- Alignment of PICTURE and VALUE clauses to highlight OCCURS and REDEFINES clauses.
- Two-way, embedded cross-references to eliminate indirect "lookups" (via a separate conventional SXREF listing).
- Reference letters to show the type of reference, indicate overall usage of a program item, and reduce the need to look up each reference.
- Footnotes on Procedure Division pages to show the definition of referenced data items, thereby eliminating more "lookups".
- Two-column Procedure Division pages to compact the listing and further reduce page turning.
- Cross-reference summary to show how, and how much, FD's and Procedure Division section's reference each other.
- Optional reformatted and renumbered source deck for manual use or for updating the BASIS library.

OVERALL OPERATION OF THE LISTER

The lister accepts source programs written in American National Standard COBOL and analyzes the source statements to establish inter-statement references, as well as the type of action resulting from the reference such as redefinition, interrogation, open/close, etc. After scanning the source statements, the lister performs all information transfers necessary for cross-referencing. Finally, the lister composes and prints the reformatted source code.

This reformatted source output follows indenting conventions imposed by the lister to increase readability, and contains cross references between data items and Procedure Division statements, between PERFORM statements and paragraph names, etc. Optionally, the lister produces a new source deck that matches the output listing except that the embedded cross-reference information is omitted.

Thus, the lister can be used to process source decks for uniformity of indenting and for highlighting of IFs, GO TOs, etc., or it can be used simply to obtain a cross-referenced source listing as permanent documentation of a production program, or as an aid in program analysis and debugging. Various options permit printing the Procedure Division listing in two columns to conserve space, and inclusion of BASIS and COPY statements.

The Listing

The reformatted output listing is divided into four parts:

1. A one-page introduction which describes briefly lister codes, conventions, uses
2. The Identification and Environment divisions
3. Detailed, cross-referenced, reformatted Data and Procedure divisions
4. The summary listing

These are described briefly below, and in greater detail in subsequent sections.

The Output Deck

The deck produced optionally by the lister may be saved either in card form or in a BASIS library. This output reflects the output listing, except that cross-reference information is omitted, and that card numbers replace statement numbers. The output deck is described in detail in a subsequent section of this chapter.

Reformatting of Identification and Environment Divisions

The lister reformats the Identification Division statements only by imposing indenting conventions. Statements are indented two spaces. Statements with continuations are indented four spaces.

Environment Division statements are reformatted by imposing indenting conventions and by appending cross-reference information to SELECT statements in the FILE CONTROL section. Thus, in reading the FILE CONTROL section, you receive direct references to the FILE DESCRIPTION statements in the Data Division.

Data Division Reformatting

The lister reformats the Data Division statements principally by imposing indenting conventions on them. In addition, it aligns PICTURE, VALUE, and other clauses vertically to improve readability and facilitate visual checking. This alignment generally highlights REDEFINES and OCCURS clauses, for example. All indenting is with respect to the left margin, which contains the statement number. The indenting conventions are:

- FDs are not indented
- For LEVEL 01 items, the indent is two spaces
- For LEVEL 02 items, indent is four spaces

Level 03 and lower items are each indented two from the last higher level item. Using this convention, the overall structure of each file and group item is immediately apparent when reading the listing. Level 77 items are not indented.

The most striking change in the appearance of the Data Division listing is the addition, at the right of each statement, of cross references that identify the statement number of each Data Division or Procedure Division statement that redefines, changes, reads, tests, or otherwise refers to the data item. When the number of such references is too great to fit on the line, the lister prints as many on the line as there is space for, and prints the remainder as a footnote at the bottom of the page.

Procedure Division Reformatting

The lister reformats the Procedure Division by applying indenting conventions to nested IFs, GO TOs, etc., and by appending cross references to sections and paragraphs, where appropriate, to indicate that the procedure is PERFORMed by another or similar action. It also appends references to the Data Division so that the data item being acted upon can be found quickly. Six codes are used in the Procedure Division:

A	ALTER
B	(ALTER) to PROCEED TO
E	INPUT or OUTPUT procedure for Sort/Merge
G	GO TO
P	PERFORM
T	(PERFORM) THRU

Summary Listing

The summary listing provides an overall view of the relationship among FDs, RDs, and SDs in the program. The entry for each of these major parts of the program consists of a title line showing the statement number and the name of the file, record, or section and a series of counts (by reference type) for each of the categories "intra", "from", and "to". Intra references are those within the section, file, or record, such as REDEFINES and PERFORM operations.

THE SOURCE LISTING

General Appearance

In looking at the source listing of the Identification, Environment, or Data Divisions, you will find that the pages may be considered as having three "columns". The leftmost contains a statement number, or is blank if the line is either a comment or a continuation of the preceding statement or line. The second column contains the reformatted COBOL statements. The third (not present in the Procedure Division) contains references to or from other statements in the source program. Thus, each line of the output listing contains a numbered source statement or its continuation, and a reference or series of references to all other statements in the source program that refer to it. If the series of references is too long to file on the line, the lister prints as many as will fit, followed by a letter indicating a footnote. The footnote contains the remainder of the references.

The source listing of the Procedure Division is normally printed in double-column format, with each column divided as described above. This format also approximately doubles the span of logic that can be seen on one page or one facing-page spread.

Another characteristic of the source listing is that regardless of whether the source code follows indentation conventions, the lister indents statements according to their type, and according to hierarchy, where applicable. This feature of the lister makes file and record structure immediately visible, and also helps to identify groups of related statements such as IF/ELSE and nesting of IFs.

Format Conventions

New statements are indented from the left margin, which contains the statement number. The lister treats as new statements

- Division headers
- Section headers
- Paragraph names
- Level numbers
- Verbs
- ELSE statements
- OTHERWISE statements
- AT END statements (only when following SEARCH statements)

Indentation of the new statement is made according to the following rules:

1. Data Division
   - FDs and Level 77 items are not indented
   - Level 01 items are indented two spaces in the FILE SECTION or REPORT SECTION and are not indented in the LINKAGE or WORKING-STORAGE sections
   - Each subsequent lower level within an 01 item is indented two spaces more than the preceding higher level
2. Procedure Division
   - Section names are not indented
   - Paragraph names are indented two spaces
   - Unconditionally-executed verbs are indented four spaces
   - Verbs executed under a single condition such as IF or AT END are indented six spaces
   - The first IF statement in a nest of IF statements is indented two spaces; subsequent nested IF statements are indented an additional two spaces at each level
   - ELSE statements are indented to the same position as the IF statement to which they refer
3. Continuation lines (in all divisions) are indented six spaces with respect to the first line of the continued statement

Word spacing within a statement and on continuation lines is usually one space. Within the Data Division, however, PICTURE and VALUE clauses are aligned as nearly as possible into columns so that they may be found and compared easily.

Words are not split at the end of a statement or continuation line unless the word to be split is a nonnumeric literal that will not fit on a single continuation line.

References appear to the right of the statement or continuation line. References following paragraph names appear immediately to the right of the name, separated by a blank. References following other types of statements appear as far to the right as possible depending on the number of blanks available on the line. Each reference consists of a statement number and a type indicator. References in series are separated by commas, and are in ascending order.

Within the Data Division, a reference may also be an alphabetic footnote indicator. The footnote contains a series of references to REDEFINES and Procedure Division statements that refer to that data item.

Within the Procedure Division, the reference may also be a footnote indicator, but the footnote is different in appearance. In the Procedure Division, the footnote is actually an on-page replica of the Data Division statement referred to by the footnoted statement. This replica is complete with all other references to the data item from other portions of the program. To conserve space in the listing, the lister does not repeat a footnote if it appears at the bottom of either of the two preceding pages.

Type Indicators

As mentioned above, a reference consists of a statement number and a type indicator. The type indicator provides immediate information as to what is being done by the statement referred to.

Two sets of type indicators are used by the lister, one for the Data Division, and one for the Procedure Division. Within the Data Division, the type indicators are:

U	Data item unchanged (used as a source field)
C	Data item changed (such as ADD or MOVE)
E	Data item referred to by Environment Division statement (SELECT) or by Procedure Division input/output operation (READ, WRITE)
D	Data item REDEFINED or RENAMEd
Q	Queried by IF, WHEN, or UNTIL
R	Referred to by a READ statement
W	Referred to by a WRITE, GENERATE, DISPLAY, or similar statement
X	Used as an index, subscript, or object of a DEPENDING ON statement

Within the Procedure Division, the type indicators are:

A	ALTER
B	(ALTER) TO PROCEED TO
E	INPUT or OUTPUT procedure (Sort or Merge feature)
G	GO TO
P	PERFORM
T	(PERFORM) THRU

THE SUMMARY LISTING

The summary listing is useful both as an analysis and as a troubleshooting aid. Using the summary listing, the data areas most referred to, the procedures that reference them most often and the nature of those references can be ascertained quickly. The number of references to undefined symbols and the number of incorrectly coded COBOL words can also be ascertained.

General Appearance

Each division or section header, and each FD, RD, or SD begins a new entry in the summary listing. The entry consists of the header line and, beginning on the next line, the total number of each kind of reference to that section from within itself (INTRA), and from outside itself (FROM). These references are followed by similar information for references the section makes to others outside itself (TO).

THE OUTPUT DECK

By specifying the DECK option on the LST card, a new COBOL source deck can be produced that reflects the reformatted source listing. This deck may be saved in a BASIS library (used directly as input to the compiler) or punched onto cards. Because of reformatting, the new deck may contain more cards than the original, but the difference is not great enough to cause any appreciable increase in compilation time. The output deck differs from the listing as follows:

1. References, footnotes, and blank lines are omitted.

2. Literals will be repositioned, if needed, to assure proper continuation.

3. Statement numbers are converted to card numbers.

   a. The statement number is multiplied by 10, and leading zeros are added as necessary to fill columns 1 through 6.

   b. Comment and continuation cards are numbered one higher than the preceding card.

   c. Statement-beginning cards are given the higher of the two numbers produced by the first two rules.

The new deck will permanently process all the BASIS INSERT and DELETE cards, and thus can be used to permanently update the Source Statement Library. This avoids having to resequence the update cards after they have been tested, and avoids the errors incurred during that resequencing process.

USING THE LISTER

Options

The format and contents of the listings and deck produced by the lister are determined by the options specified on the LST card. The LIST card may be placed anywhere between the EXEC statement and the first statement of the COBOL program. It may be placed between any other compiler option cards.

Two format options determine the dimensions and layout of the source and summary listings.

PROC=<u>1col</u>
2col
: specifies that the source listing of the Procedure Division will be printed in either single or double column format. At least 132 print positions are required for double column format.

Three options pertain to the output deck:

DECK
<u>NODECK</u>
: indicates whether an updated source deck is to be produced as a result of the lister reformatting and/or the update basis library.

COPYPCH
<u>NOCOPYPCH</u>
: will punch updated and reformatted copy libraries as a permanent part of the source when DECK is specified, and will punch out an updated and reformatted copy library when no updated source deck is requested.

<u>LSTONLY</u>
LSTCOMP
: The LSTONLY option will give a reformatted listing and a deck, if DECK was specified, but will not compile the program. LISTCOMP will, in addition to listing the source, also compile the program as part of the job step.

PROGRAMMING CONSIDERATIONS

The lister is designed to operate most efficiently on syntactically correct COBOL source, and does not have the expanded error handling of the full compiler. It is therefore highly recommended that the user programs first be compiled using the SYNTAX option, and syntax errors corrected before invoking the lister feature. If the lister function is used and there are syntactical errors, the formatting may be unpredictable, and performance can be significantly impacted.

Unusual termination of lister can occur if the source program contains:

- Too many (approximately 80 or more) consecutive (*) comment cards.

- Too many (approximately 100 or more) consecutive blank cards.

<u>Further notes</u>: Since Lister reformats the users COBOL program, compilation of the program, if LSTCOMP is in effect, will be different from a non-lister compilation of the same program. For example:

1. Lister sequence numbers may be different

2. SKIP/EJECT cards will have no functional value with LISTER

3. BASIS card will be dropped from the Lister listings

4. FIPS messages will be based on the reformatted Lister listings.

5. Suppress option of COPY will have no effect

6. Sequence checking will not take place for a Lister sum.

7. The Insert card indicator for BASIS will not be indicated on a lister listing.

A programmer using IBM DOS/VS COBOL under the DOS/VS System, has several methods available to him for testing and debugging his programs. Use of the symbolic debugging features is the easiest and most efficient method for testing and debugging and is described in detail in this chapter.

The chapter entitled "Program Checkout" contains information useful for testing and debugging programs run without the symbolic debugging features. It also contains information on compile-time debugging features, linkage editor and execution-time diagnostics as well as a description of taking checkpoints and restarting programs.

The chapter entitled "Execution Statistics" also contains information helpful in testing and debugging programs run both with and without the symbolic debugging features.

USE OF THE SYMBOLIC DEBUGGING FEATURES

There are three symbolic debugging options available to the programmer for object-time debugging: the statement number option, the flow trace option, and the symbolic debugging option. None of these features require source language coding; rather they are requested via the CBL card at compile time. Operation of the symbolic debug option is dependent upon execution-time control cards. Figure 9 illustrates the output generated for each of these features.

STATEMENT NUMBER OPTION

The statement number option facilitates debugging by providing the programmer with information about the statement being executed at the time of an abnormal termination of a job. It identifies the program containing the statement and provides the number of the statement and of the verb being executed.

This feature is requested at compile time via the STATE option of the CBL card. Note that STATE and STXIT, STATE and SYMDMP, and STATE and OPT are mutually exclusive options at compile-time and STATE and STXIT are mutually exclusive in an execution-time run unit. The CBL card is discussed in detail in the chapter "Preparing COBOL Programs for Processing."

FLOW TRACE OPTION

The flow trace option provides the programmer with the facility for receiving a formatted trace (i.e., a list containing the program identification and statement numbers) corresponding to a variable number of procedures executed prior to an abnormal termination. The number of procedures to be traced is specified by the programmer. If the FLOW option is specified and the number of procedures is not specified, a trace of 99 procedures is provided.

A flow trace is printed only in the event of an abnormal termination. It is requested at compile time via the FLOW option of the CBL card. In a subprogram structure, once a FLOW specification has been made on a program, the subprograms for which a trace is desired should specify FLOW=0. The FLOW=0 specification enables subprograms to utilize the table space reserved previously for the trace; additional table space need not be allocated.

FLOW and STXIT, and FLOW and OPT are mutually exclusive options at compile-time and FLOW and STXIT are mutually exclusive in an execution-time run unit. The CBL card is discussed in the chapter "Preparing COBOL Programs for Processing."

SYMBOLIC DEBUG OPTION

The symbolic debug option produces a symbolic formatted dump of the object program's data area when the program abnormally terminates. It also enables the programmer to request dynamic dumps of specific data-names at strategic points during program execution. If two or more COBOL programs are link edited together and one of them terminates abnormally, the program causing termination and any callers compiled with the symbolic debug option, up to and including the main program, will be given a formatted dump. If any called program contains the SYMDMP option, the main program must be an ANS COBOL program.

Another feature of SYMDMP is that a check is made for a subscript which points out of the program area and for the length of a variable-length move out of the data area. If these address limits are reached, message C170I is issued and an abend dump is given.

The abnormal termination dump consists of the following parts:

1. Abnormal termination message, including the number of the statement and of the verb being executed at the time of an abnormal termination.

2. Selected areas in the Task Global Table.

3. Formatted dump of the Data Division including:

   (a) for an SD, the statement number, the sort-file-name, the type, and the sort record.

   (b) for an FD, the statement number, the file-name, the type, SYSnnn, DTF status, the contents of the Pre-DTF and DTF in hexadecimal, and the fields of the record.

   for a VSAM file, the file-name, whether the file is open or closed, file organization, type of access, type of last input-output statement, the current contents of the FILE STATUS word, as well as the record fields.

   (c) for an RD, the statement number, the report-name, the type, the report line, and the contents of PAGE-COUNTER and LINE-COUNTER if present.

   (d) for an index-name, the name, the type, and the occurrence number in decimal.

   Note: For DTFDA when ACCESS IS RANDOM, the actual key is not provided in the Pre-DTF.

The symbolic debug option is requested at compile time via the SYMDMP option of the CBL card. Note that SYMDMP and STXIT, SYMDMP and STATE, and SYMDMP and OPT are mutually exclusive options at compile time and SYMDMP and STATE and STXIT are mutually exclusive in a single execution-time run unit. The CBL card is discussed in the chapter "Preparing COBOL Programs for Processing."

Operation of the symbolic debug option is dependent on object-time control cards placed in the input stream. These cards are discussed below.

Object-Time Control Cards

The operation of the symbolic debug option is determined by two types of control cards:

Program-control card -- required if abnormal termination and/or dynamic dumps are requested.

Line-control card -- required only if dynamic dumps are requested.

Syntax Rules: The fields of both the program-control card and the line-control card must conform to the following rules:

1. Control cards are essentially free form, i.e., parameters coded on these cards can start in any column. However, parameters may not extend beyond column 71.

2. Each parameter except the last must be immediately followed by a comma.

3. No commas are needed to account for optional parameters that are not specified.

4. All upper-case letters represent specifications that are to appear in the actual statement exactly as shown.

5. All lower-case letters represent generic terms that are to be replaced in the actual statement.

6. Brackets are used to indicate that a specification is optional and is not always required in the statement.

7. Brackets enclosing stacked items indicate that a choice of one item may, but need not, be made by the programmer.

8. Braces enclosing stacked items indicate that a choice of one item must be made by the programmer.

9. All punctuation marks and special characters shown in the statement formats other than hyphens, brackets, braces, and underscores, must be punched exactly as shown. This includes commas, parentheses, and the equal sign.

10. Underscoring indicates the default case.

Continuation Cards: To continue either the program-control card or the line-control card, a nonblank character must be coded in column 72 of the continued card.

Individual keywords and data-names cannot be split between cards.

Control Statement Placement: The placement of the control cards in the input stream must be as follows:

1. If a main program is compiled with the SYMDMP option, the control cards must precede the programmer's data, if any, in the input stream:

```
// EXEC
   {Control Cards}
/*

   {Programmer's Data}
/*
/&
```

   If the main program is compiled without the SYMDMP option, but at least one subprogram has been compiled with the SYMDMP option, then two alternatives exist:

   a. If all data card files have reached EOF before the subroutine compiled with SYMDMP is called, then the following sequence should be used:

```
// EXEC
   {Programmer's Data for
     Main Program}
/*
   {Control Cards}
/*
/&
```

   b. If calls to the subroutine compiled with SYMDMP are interspersed with reading of card files, then a dummy subroutine, consisting of only an EXIT PROGRAM statement and compiled with the SYMDMP option, should be called as the first statement of the main program. The placement of control cards is as follows:

```
// EXEC
   {Control Cards}
/*
   {Programmer's Data}
/*
/&
```

Program-Control Cards: A program-control card must be present at execution time for any program requesting a SYMDMP service. Program-control cards have the following format:

```
program-id,nnn

   ,SD[=filename]    ENTRY      ,(HEX)
   ,MT[=filename]    NOENTRY    ,(NOHEX)
```

program-id
: is a one through eight character COBOL program-name. This program-name must be the name of a COBOL program compiled with the SYMDMP option. This parameter is required and must appear first on the program-control card.

nnn
: is a 3-digit integer representing the programmer logical unit assigned to the dictionary file produced at compile time (i.e., the SYS005 file.) This parameter is required and must follow the "program-id". This value must be the same as the one specified in the ASSGN control statement for the dictionary file at execution time.

SD[=filename]
MT[=filename]
: SD must be specified if the symbolic unit indicated by "nnn" is a disk file; MT must be specified if it is a tape file. "filename" is the name of the dictionary file produced at compile time. For a tape file, the "filename" parameter is ignored. For a disk file, if "filename" is not specified, IJSYS05 will be used. "filename" may be from one to seven characters in length. If "filename" is specified on the CBL card for a disk file, "filename" must also be specified on the program-control card and these names must be identical.

ENTRY
NOENTRY
: ENTRY is used to provide a trace of a program-name when several programs are link edited together. Each time the program whose PROGRAM-ID matches the "program-id" parameter is entered, its name is displayed.

(HEX)
(NOHEX)
: refers to the format of the Data Division area provided in the abnormal termination dump. If HEX is specified, level-01 items are provided in hexadecimal. Items subordinate to level-01 items are printed in EBCDIC, if possible. Level-77 items are provided both in EBCDIC and hexadecimal. If HEX is not specified, items subordinate to level-01 items and level-77 items are provided in EBCDIC. If unprintable, hexadecimal notation is provided.

Note: Parentheses are required.

Line-Control Cards: Line-control cards have the following format:

```
line-num [,(verb-num)][,ON n][,m][,k]

⎧ ⎡,(HEX)  ⎤                              ⎫
⎪ ⎣,(NOHEX)⎦  ,ALL                        ⎪
⎨                                         ⎬
⎪ ⎧⎡,(HEX)  ⎤                          ⎫  ⎪
⎩ ⎩⎣,(NOHEX)⎦,name1 [THRU name2]...⎭  ⎭
```

line-num
: corresponds to the generated card number prior to which the dump is desired. The dump is given before the first or only verb on that line. This parameter is required and must be the first on the line-control card.

verb-num
: indicates the position of the verb on the specified statement before whose execution a dynamic dump is given. When "verb-num" is not specified, 1 is assumed; when specified, "verb-num" must follow line-num and may not exceed 15.

ON n [,m][,k]
: is equivalent to the COBOL statement ON n AND EVERY m UNTIL k. This option limits the requested dynamic dumps to specified times. For example, "ON n" would result in one dump, given the nth time "line-num" is reached during execution. "ON n,m" would result in a dump the first time at the nth execution of "line-num" and thereafter at every mth execution until end-of-job.

(HEX)
(NOHEX)
: refers to the format of the Data Division areas provided in the dynamic dump. If HEX is specified, level-01 items are provided in hexadecimal. Items subordinate to level-01 items are printed in EBCDIC, if possible. Level-77 items are printed both in EBCDIC and hexadecimal. If HEX is not specified, items subordinate to level-01 items and level-77 items are provided in EBCDIC. If unprintable, hexadecimal notation is provided. Note that if "name1" is specified and it represents a group item and HEX has not been specified, neither the group nor the elementary items in the group will be provided in hexadecimal.

name1 [THRU name2]
: represents selected areas of the Data Division to be dumped. With the THRU option, a range of data-names appearing consecutively in the Data Division is dumped. "name1" and "name2" may be qualified but not subscripted. If the programmer wishes to see a subscripted item, specifying the name of the item without the subscript results in a dump of of every occurrence of that item.

ALL
: results in a dump of everything that would be dumped in the event of an abnormal termination. The purpose of ALL is to allow the programmer to receive a formatted dump at normal end-of-job. To do this, the generated statement number of the line on which a STOP RUN, EXIT PROGRAM, or GOBACK statement appears must be specified as the "line-num" parameter.

OVERALL CONSIDERATIONS

The end-of-file control card, slash asterisk (/*) must end the symbolic debug control card data set. If a run unit includes one or more programs that have been compiled with the SYMDMP option and no symbolic dump is required at execution time, the input data set must nevertheless be provided, although in this case it consists only of the end-of-file (/*) card.

If no executable output is produced as a result of the compilation (NOLINK, NODECK), any symbolic debugging options specified are suppressed.

SAMPLE PROGRAM -- TESTRUN

Figure 64 is an illustration of a program that utilizes the symbolic debugging features. In the following description of the program and its output, letters identifying the text correspond to letters in the program listing.

(A) Because the SYMDMP option is requested on the CBL card, the logical unit SYS005 must be assigned at compile time.

(B) The CBL card specifications indicate that an alphabetically ordered cross-reference dictionary, a flow trace of 10 procedures, and the symbolic debug option are being requested.

Ⓒ An alphabetically ordered cross-reference dictionary of data-names and procedure-names is produced by the compiler as a result of the SXREF specification on the CBL card.

Ⓓ The file assigned at compile time to SYS005 to store SYMDMP information is assigned to SYS009 at execution time.

Ⓔ The SYMDMP control cards placed in the input stream at execution time are printed along with any diagnostics.

① The first card is the program-control card where:

(a) TESTRUN is the PROGRAM-ID.
(b) 9 is the logical unit to which the SYMDMP file is assigned.
(c) MT indicates that the SYMDMP file is on tape.
(d) (HEX) indicates the format of the abnormal termination dump.

(2) The second card is a line-control card which requests a (HEX) formatted dynamic dump of KOUNT, NAME-FIELD, NO-OF-DEPENDENTS, and RECORD-NO prior to the first and every fourth execution of generated card number 71.

(3) The third card is also a line-control card which requests a (HEX) formatted dynamic dump of WORK-RECORD and B prior to the execution of generated card number 80.

(F) The type code combinations used to identify data-names in abnormal termination and dynamic dumps are defined. Individual codes are illustrated in Table 34.

(G) The dynamic dumps requested by the first line-control card.

(H) The dynamic dumps requested by the second line-control card.

(I) Program interrupt information is provided by the system when a program terminates abnormally.

(J) The statement number information indicates the number of the verb and of the statement being executed at the time of the abnormal termination. The name of the program containing the statement is also provided.

(K) A flow trace of the last 10 procedures executed is provided because FLOW=10 was specified on the CBL card.

(L) Selected areas of the Task Global Table are provided as part of the abnormal termination dump.

(M) For each file-name, the generated card number, the file type, SYSnnn, the DTF status, and the fields of the Pre-DTF and DTF in hexadecimal are provided.

(N) The fields of records associated with each FD are provided in the format requested on the program-control card.

(P) The contents of the fields of the Working-Storage Section are provided in the format requested on the program-control card.

(Q) The value associated with each of the possible subscripts are provided for data items described with an OCCURS clause.

(R) Asterisks appearing within the EBCDIC representation of the value of a given field indicate that the type and the actual content of the field conflict.

Note: When using the SYMDMP option, level numbers appear "normalized" in the symbolic dump produced. For example, a group of data items described as:

```
   01  RECORDA.
       05 FIELD-A.
          10  FIELD-A1 PIC X.
          10  FIELD-A2 PIC X.
```

will appear as follows in SYMDMP output:

```
   01 RECORDA...
   02 FIELD-A...
   03 FIELD-A1...
   03 FIELD-A2...
```

Debugging TESTRUN

1. Referring to the statement number information (J) provided by the symbolic debug option, it is learned that the abend occurred during the execution of the first verb on card 80.

2. Generated card number 80 contains the statement COMPUTE B = B + 1.

3. Verifying the contents of B at the time of the abnormal termination (R) it can be seen that the usage of B (numeric packed) conflicts with the value contained in the data area reserved for B (numeric display).

4. The abnormal termination occurred while trying to perform an addition on a display item.

More complex errors may require the use of dynamic dumps to isolate the problem area. Line-control cards are included in TESTRUN merely to illustrate how they are used and the output they produce.

Table 34. Individual Type Codes Used in SYMDMP Output

Code	Meaning
A	Alphabetic
B	Binary
D	Display
E	Edited
*	Subscripted Item
F	Floating Point
N	Numeric
P	Packed Decimal
S	Signed
OT	Overpunch Sign Trailing
OL	Overpunch Sign Leading
SL	Separate Sign Leading
ST	Separate Sign Trailing

```
// JOB DEBUGL
// OPTION NODECK,LINK,LIST,LISTX,SYM,ERRS
// ASSGN SYS005,X'483'  <-- A
// EXEC FCOBOL
```

```
     1  IBM DOS VS COBOL                              REL 1.0          PP NO. 5746-CB1                07.52.05  10/02/73

  CBL SXREF,FLOW=10,SYMDMP,QUOTE,SEQ  <-- B
00001    000010 IDENTIFICATION DIVISION.
00002    000020 PROGRAM-ID. TESTRUN.
00003    000030     AUTHOR. PROGRAMMER NAME.
00004                  INSTALLATION.  NEW YORK DEVELOPMENT CENTER.
00005                  DATE-WRITTEN.  SEPTEMBER 26,1973.
00006           DATE-COMPILED. 10/02/73
00007           REMARKS.  THIS PROGRAM HAS BEEN WRITTEN AS A SAMPLE PROGRAM FOR
00008               COBOL USERS.  IT CREATES AN OUTPUT FILE AND READS IT BACK
00009               AS INPUT.
00010    000100
00011    000110 ENVIRONMENT DIVISION.
00012    000120 CONFIGURATION SECTION.
00013    000130 SOURCE-COMPUTER. IBM-360-H50.
00014    000140 OBJECT-COMPUTER. IBM-370.
00015    000150 INPUT-OUTPUT SECTION.
00016    000160 FILE-CONTROL.
00017    000170     SELECT FILE-1 ASSIGN TO SYS008-UT-2400-S.
00018    000180     SELECT FILE-2 ASSIGN TO SYS008-UT-2400-S.
00019    000190
00020    000200 DATA DIVISION.
00021    000210 FILE SECTION.
00022    000220 FD  FILE-1
00023    000230     LABEL RECORDS ARE OMITTED
00024    000240     BLOCK CONTAINS 5 RECORDS
00025    000250     RECORDING MODE IS F
00026    000255     RECORD CONTAINS 20 CHARACTERS
00027    000260     DATA RECORD IS RECORD-1.
00028    000270 01  RECORD-1.
00029               05  FIELD-A PIC X(20).
00030    000290 FD  FILE-2
00031    000300     LABEL RECORDS ARE OMITTED
00032    000310     BLOCK CONTAINS 5 RECORDS
00033    000320     RECORD CONTAINS 20 CHARACTERS
00034    000330     RECORDING MODE IS F
00035    000340     DATA RECORD IS RECORD-2.
00036    000350 01  RECORD-2.
00037               05  FIELD-A PIC X(20).
```

Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN (Part 1 of 12)

```
        IBM DOS VS COBOL                          REL 1.0          PP NO. 5746-CB1                 07.52.05  10/02/73

C0038   000370 WORKING-STORAGE SECTION.
C0039   000380 01  FILLER.
C0040              02  KOUNT PIC S99 COMP SYNC.
00041              02  ALPHABET PIC X(26) VALUE "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
00042              02  ALPHA REDEFINES ALPHABET PIC X OCCURS 26 TIMES.
C0043   000420     02  NUMBR PIC S99 COMP SYNC.
C0044              02  DEPENDENTS PIC X(26) VALUE "01234012340123401234012340".
C0045              02  DEPEND REDEFINES DEPENDENTS PIC X OCCURS 26 TIMES.
C0046   000450 01  WORK-RECORD.
00047   000460     05 NAME-FIELD PIC X.
C0048   000470     05 FILLER PIC X.
00049   000480     05 RECORD-NO PIC 9999.
C0050   000490     05 FILLER PIC X VALUE IS SPACE.
00051              05 LOCATION PIC AAA VALUE IS "NYC".
C0052   000510     05 FILLER PIC X VALUE IS SPACE.




C0053   000520     05 NO-OF-DEPENDENTS PIC XX.
C0054              05  FILLER PIC X(7) VALUE IS SPACES.
C0055          01  RECORDA.
C0056              02  A PICTURE S9(4) VALUE 1234.
00057              02  B REDEFINES A PICTURE S9(7) COMPUTATIONAL-3.
C0058   000550 PROCEDURE DIVISION.
C0059          BEGIN.
00060              NOTE THAT THE FOLLOWING OPENS THE OUTPUT FILE
C0061              TO BE CREATED AND INITIALIZES THE COUNTERS.
C0062          STEP-1.  OPEN OUTPUT FILE-1.  MOVE ZERO TO KOUNT, NUMBR.
C0063              NOTE THAT THE FOLLOWING CREATES INTERNALLY THE
C0064              RECORDS TO BE CONTAINED IN THE FILE, WRITES THEM
C0065              ON TAPE, AND DISPLAYS THEN ON THE CONSOLE.
C0066          STEP-2.  ADD 1 TO KOUNT, NUMBR.  MOVE ALPHA (KOUNT) TO
C0067              NAME-FIELD.
C0068              MOVE DEPEND (KOUNT) TO NO-OF-DEPENDENTS.
C0069   000660     MOVE NUMBR TO RECORD-NO.
C0070          STEP-3.  DISPLAY WORK-RECORD UPON CONSOLE.
00071              WRITE RECORD-1 FROM WORK-RECORD.
C0072          STEP-4.  PERFORM STEP-2 THRU STEP-3
C0073              UNTIL KOUNT IS EQUAL TO 26.
C0074              NOTE THAT THE FOLLOWING CLOSES THE OUTPUT FILE
C0075              AND REOPENS IT AS INPUT.
C0076   000720 STEP-5. CLOSE FILE-1. OPEN INPUT FILE-2.
C0077              NOTE THAT THE FOLLOWING READS BACK THE FILE
C0078              AND SINGLES OUT EMPLOYEES WITH NO DEPENDENTS.
C0079          STEP-6.  READ FILE-2 RECORD INTO WORK-RECORD AT END GO TO STEP-8.
C0080              COMPUTE B = B + 1.
C0081          STEP-7.  IF NO-OF-DEPENDENTS IS EQUAL TO "0" MOVE "Z" TO
C0082              NO-OF-DEPENDENTS.  EXHIBIT NAMED WORK-RECORD.
C0083              GO TO STEP-6.
00084   000780 STEP-8. CLOSE FILE-2.
C0085   000790     STOP RUN.
```

Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN (Part 2 of 12)

```
IBM DOS VS COBOL                       REL 1.0                PP NO. 5746-CB1                07.52.05  10/02/73


INTRNL NAME    LVL SOURCE NAME                     BASE    DISPL   INTRNL NAME    DEFINITION    USAGE        R   O   Q   M


DNM=1-148      FD  FILE-1                         DTF=01           DNM=1-148                    DTFMT                        F
DNM=1-179      01  RECORD-1                        BL=1     000    DNM=1-179      DS 0CL20      GROUP
DNM=1-200      02  FIELD-A                         BL=1     000    DNM=1-200      DS 20C        DISP
DNM=1-217      FD  FILE-2                         DTF=02           DNM=1-217                    DTFMT                        F
DNM=1-248      01  RECORD-2                        BL=2     000    DNM=1-248      DS 0CL20      GROUP
DNM=1-269      02  FIELD-A                         BL=2     000    DNM=1-269      DS 20C        DISP
DNM=1-289      01  FILLER                          BL=3     000    DNM=1-289      DS 0CL56      GROUP
DNM=1-308      02  KOUNT                           BL=3     000    DNM=1-308      DS 1H         COMP
DNM=1-323      02  ALPHABET                        BL=3     002    DNM=1-323      DS 26C        DISP
DNM=1-341      02  ALPHA                           BL=3     002    DNM=1-341      DS 1C         DISP         R   O
DNM=1-359      02  NUMBER                          BL=3     01C    DNM=1-359      DS 1H         COMP
DNM=1-374      02  DEPENDENTS                      BL=3     01E    DNM=1-374      DS 26C        DISP
DNM=1-394      02  DEPEND                          BL=3     01E    DNM=1-394      DS 1C         DISP         R   O
DNM=1-410      01  WORK-RECORD                     BL=3     038    DNM=1-410      DS 0CL20      GROUP
DNM=1-434      02  NAME-FIELD                      BL=3     038    DNM=1-434      DS 1C         DISP
DNM=1-454      02  FILLER                          BL=3     039    DNM=1-454      DS 1C         DISP
DNM=1-473      02  RECORD-NO                       BL=3     03A    DNM=1-473      DS 4C         DISP-NM
DNM=1-492      02  FILLER                          BL=3     03E    DNM=1-492      DS 1C         DISP
DNM=2-000      02  LOCATION                        BL=3     03F    DNM=2-000      DS 3C         DISP
DNM=2-018      02  FILLER                          BL=3     042    DNM=2-018      DS 1C         DISP
DNM=2-037      02  NO-OF-DEPENDENTS                BL=3     043    DNM=2-037      DS 2C         DISP
DNM=2-063      02  FILLER                          BL=3     045    DNM=2-063      DS 7C         DISP
DNM=2-082      01  RECORDA                         BL=3     050    DNM=2-082      DS 0CL4       GROUP
DNM=2-102      02  A                               BL=3     050    DNM=2-102      DS 4C         DISP-NM
DNM=2-113      02  B                               BL=3     050    DNM=2-113      DS 4P         COMP-3       R
```

Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN (Part 3 of 12)

```
IBM DOS VS COBOL                                    REL 1.0           PP NO. 5746-CB1               07.52.05  10/02/73


                 MEMORY MAP

             TGT                    00400

          SAVE AREA                 00400
          SWITCH                    00448
          TALLY                     0044C
          SORT SAVE                 00450
          ENTRY-SAVE                00454
          SORT CORE SIZE            00458
          NSTD-REELS                0045C
          SORT RET                  0045E
          WORKING CELLS             00460
          SORT FILE SIZE            00590
          SORT MODE SIZE            00594
          PGT-VN TBL                00598
          TGT-VN TBL                0059C
          SORTAB ADDRESS            005A0
          LENGTH OF VN TBL          005A4
          LNGTH OF SORTAB           005A6
          PGM ID                    005A8
          A(INIT1)                  005B0
          UPSI SWITCHES             005B4
          DEBUG TABLE PTR           005BC
          CURRENT PRIORITY          005C0
          TA LENGTH                 005C1
          PROCEDURE BLOCK1 PTR      005C4
          UNUSED                    005C8
          RESERVED                  005CC
          VSAM SAVE AREA ADDRESS    005D0
          UNUSED                    005D4
          RESERVED                  005DC
          OVERFLOW CELLS            005F4
          BL CELLS                  005F4
          DTFADR CELLS              00600
          FIB CELLS                 00608
          TEMP STORAGE              00608
          TEMP STORAGE-2            00610
          TEMP STORAGE-3            00610
          TEMP STORAGE-4            00610
          BLL CELLS                 00610
          VLC CELLS                 00614
          SBL CELLS                 00614
          INDEX CELLS               00614
          SUBADR CELLS              00614
          ONCTL CELLS               0061C
          PFMCTL CELLS              0061C
          PFMSAV CELLS              0061C
          VN CELLS                  00620
          SAVE AREA =2              00624
          XSASW CELLS               00624
          XSA CELLS                 00624
          PARAM CELLS               00624
          RPTSAV AREA               00628
          CHECKPT CTR               00628
          IOPTR CELLS               00628
          DEBUG TABLE               00628
```

Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN (Part 4 of 12)

```
     IBM DOS VS COBOL                           REL 1.0            PP NO. 5746-CBl              07.52.05  10/02/73

  LITERAL POOL (HEX)

  00690 (LIT+0)      00000001  001A1C5B  5BC2D6D7  C5D5405B  5BC2C3D3  D6E2C55B
  006A8 (LIT+24)     5BC2C6C3  D4E4D300  C0000000

       DISPLAY LITERALS (BCD)

  006B4 (LTL+36)    'WORK-RECORD'

                      PGT                      00638

                 DEBUG LINKAGE AREA            00638
                 OVERFLOW CELLS                00640
                 VIRTUAL CELLS                 00640
                 PROCEDURE NAME CELLS          00664
                 GENERATED NAME CELLS          00674
                 SUBDTF ADDRESS CELLS          00688
                 VNI CELLS                     00688
                 LITERALS                      00690
                 DISPLAY LITERALS              006B4
                 PROCEDURE BLOCK CELLS         006C0

   REGISTER ASSIGNMENT

    REG 6    BL =3
    REG 7    BL =1
    REG 8    BL =2

 WORKING-STORAGE STARTS AT LOCATION 00100 FOR A LENGTH OF 00058.

  0
  59
                        0006C0                        START     EQU    *
                        0006C0  58 F0 C 018                     L      15,018(0,12)           V(ILBDDBG4)
                        0006C4  05 EF                           BALR   14,15
                        0006C6  58 F0 C 01C                     L      15,01C(0,12)           V(ILBDFLW1)
                        0006CA  05 1F                           BALR   1,15
                        0006CC  003B                            DC     X'003B'
  62
                        0006CE  58 F0 C 018                     L      15,018(0,12)           V(ILBDDBG4)
                        0006D2  05 EF                           BALR   14,15
                        0006D4  58 F0 C 01C                     L      15,01C(0,12)           V(ILBDFLW1)
                        0006D8  05 1F                           BALR   1,15
                        0006DA  003E                            DC     X'003E'
  62
                        0006DC  58 F0 C 018                     L      15,018(0,12)           V(ILBDDBG4)
                        0006E0  05 EF                           BALR   14,15
                        0006E2  58 20 D 1F4                     L      2,1F4(0,13)            BL =1
                        0006E6  41 10 C 05F                     LA     1,05F(0,12)            LIT+7
                        0006EA  58 00 D 200                     L      0,200(0,13)            DTF=1
                        0006EE  18 40                           LR     4,0
                        0006F0  07 00                           BCR    0,0
                        0006F2  05 F0                           BALR   15,0
                        0006F4  50 00 F 008                     ST     0,008(0,15)


 *STATISTICS*            SOURCE RECORDS =      85      DATA ITEMS =      25      NO OF VERBS =      29
 *STATISTICS*            PARTITION SIZE = 655176       LINE COUNT =      56      BUFFER SIZE =     512
 *OPTIONS IN EFFECT*     PMAP RELOC ADR =    NONE      SPACING    =       1      FLOW        =      10
 *OPTIONS IN EFFECT*        LISTX        QUOTE         SYM     NOCATALR       LIST          LINK      NOSTXIT        NOLIB
 *OPTIONS IN EFFECT*      NOCLIST        FLAGW         ZWB     NOSUPMAP      NOXREF         ERRS        SXREF        NOOPT
 *OPTIONS IN EFFECT*      NOSTATE        TRUNC         SEQ       SYMDMP      NODECK        NOVERB     NOSYNTAX        NOLVL
```

Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN (Part 5 of 12)

```
IBM DOS VS COBOL                               REL 1.0          PP NO. 5746-CB1                    07.52.05  10/02/73

                                          CROSS-REFERENCE DICTIONARY
  (C)
DATA NAMES                           DEFN      REFERENCE


A                                    000056
ALPHA                                000042    000066
ALPHABET                             000041
B                                    000057    000080
DEPEND                               000045    000068
DEPENDENTS                           000044
FIELD-A                              000029
FIELD-A                              000037
FILE-1                               000017    000062   000071   000076
FILE-2                               000018    000076   000079   000084
KOUNT                                000040    000062   000066   000068   000072
LOCATION                             000051
NAME-FIELD                           000047    000066
NO-OF-DEPENDENTS                     000053    000068   000081
NUMBR                                000043    000062   000066   000069
RECORD-NO                            000049    000069
RECORD-1                             000028    000071
RECORD-2                             000036    000079
RECORDA                              000055
WORK-RECORD                          000046    000070   000071   000079   000082
BEGIN                                000059
STEP-1                               000062
STEP-2                               000066    000072
STEP-3                               000070    000072
STEP-4                               000072
STEP-5                               000076
STEP-6                               000079    000083
STEP-7                               000081
STEP-8                               000084    000079



CARD   ERROR MESSAGE

00056  ILA2190I-W     PICTURE CLAUSE IS SIGNED, VALUE CLAUSE UNSIGNED. ASSUMED POSITIVE.
00066  ILA5011I-W     HIGH ORDER TRUNCATION MIGHT OCCUR.
00066  ILA5011I-W     HIGH ORDER TRUNCATION MIGHT OCCUR.


// EXEC LNKEDT



JOB  DEBUGL                DOS LINKAGE EDITOR DIAGNOSTIC OF INPUT

ACTION TAKEN  MAP REL
LIST   AUTOLINK   IJFFBZZN
LIST   AUTOLINK   ILBDADR0
LIST   AUTOLINK   ILBDDBG0
LIST   INCLUDE IJJCPDV
LIST   AUTOLINK   ILBDDSP0
LIST   AUTOLINK   ILBDDSS0
LIST    INCLUDE IJJCPDV
LIST   AUTOLINK   ILBDFLW0
LIST   AUTOLINK   ILBDIML0
LIST   AUTOLINK   ILBDMNS0
LIST   AUTOLINK   ILBDSAE0
LIST   ENTRY
```

Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN (Part 6 of 12)

```
IBM DOS VS COBOL                          REL 1.0            PP NO. 5746-CB1                07.52.05  10/02/73


     PHASE  XFR-AD  LOCORE  HICORE  DSK-AD   ESD TYPE  LABEL     LOADED  REL-FR

  PHASE***  07D878  07D878  0803EF  05F 0F 4  CSECT     TESTRUN   07D878  07D878  RELOCATABLE

                                              CSECT     IJFFBZZN  07E368  07E368
                                              *  ENTRY  IJFFZZZN  07E368
                                              *  ENTRY  IJFFBZZZ  07E368
                                              *  ENTRY  IJFFZZZZ  07E368

                                              CSECT     ILBDSAE0  080268  080268
                                                 ENTRY  ILBDSAE1  0802B0

                                              CSECT     ILBDMNS0  080260  080260

                                              CSECT     ILBDDBG0  07EAC0  07EAC0
                                                 ENTRY  ILBDDBG5  07EFA2
                                                 ENTRY  ILBDDBG4  07F014
                                                 ENTRY  ILBDDBG7  07F038
                                                 ENTRY  ILBDDBG2  07ED8A
                                              *  ENTRY  ILBDDBG1  07EC1C
                                              *  ENTRY  ILBDDBG3  07F00A
                                              *  ENTRY  ILBDDBG6  07F024
                                                 ENTRY  STXITPSW  07F0D0
                                              *  ENTRY  SORTEP    07F270

                                              CSECT     ILBDFLW0  07FD70  07FD70
                                                 ENTRY  ILBDFLW1  07FE30
                                                 ENTRY  ILBDFLW2  07FF0C

                                              CSECT     ILBDIML0  080208  080208

                                              CSECT     ILBDADR0  07E718  07E718
                                              *  ENTRY  ILBDADR1  07E724

                                              CSECT     ILBDDSP0  07F518  07F518
                                                 ENTRY  ILBDDSP1  07F918

                                              CSECT     IJJCPDV   07E878  07E878
                                                 ENTRY  IJJCPDV1  07E878
                                              *  ENTRY  IJJCPDV2  07E878

                                              CSECT     ILBDDSS0  07FA48  07FA48
                                                 ENTRY  ILBDDSS1  07FCA8
                                                 ENTRY  ILBDDSS2  07FCA0
                                                 ENTRY  ILBDDSS3  07FD60
                                                 ENTRY  ILBDDSS4  07FA6E
                                                 ENTRY  ILBDDSS5  07FB1A
                                                 ENTRY  ILBDDSS6  07FB7A
                                                 ENTRY  ILBDDSS7  07FB44
                                                 ENTRY  ILBDDSS8  07FA9E

* UNREFERENCED SYMBOLS                        WXTRN     ILBDSTN0
                                              WXTRN     ILBDSRT0
                                              WXTRN     ILBDTEF3

003 UNRESOLVED ADDRESS CONSTANTS


// ASSGN SYS008,X'482'
// ASSGN SYS009,X'483' ←——(D)
// EXEC
```

Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN (Part 7 of 12)

```
    IBM DOS VS COBOL                          REL 1.0         PP NO. 5746-CB1          07.52.05  10/02/73

                                                  SYMDMP CONTROL CARDS


   ① TESTRUN,009,MT,(HEX)                                                          )
   ② 71,ON 1,4,(HEX),KOUNT,NAME-FIELD,NO-OF-DEPENDENTS,RECORD-NO                   }  Ⓔ
   ③ 80,(HEX),WORK-RECORD,B                                                        )

                                            NO ERRORS FOUND IN CONTROL CARDS



                                                          Ⓕ

                                            TYPE CODES USED IN SYMDMP OUTPUT

                                    CODE      MEANING

                                     A      = ALPHABETIC
                                     AN     = ALPHANUMERIC
                                     ANE    = ALPHANUMERIC EDITED
                                     D      = DISPLAY (STERLING NONREPORT)
                                     DE     = DISPLAY EDITED (STERLING REPORT)
                                     F      = FLOATING POINT (COMP-1/COMP-2)
                                     FD     = FLOATING POINT DISPLAY (EXTERNAL FLOATING POINT)
                                     NB     = NUMERIC BINARY UNSIGNED (COMP)
                                     NB-S   = NUMERIC BINARY SIGNED
                                     ND     = NUMERIC DISPLAY UNSIGNED (EXTERNAL DECIMAL)
                                     ND-OL  = NUMERIC DISPLAY OVERPUNCH SIGN LEADING
                                     ND-OT  = NUMERIC DISPLAY OVERPUNCH SIGN TRAILING
                                     ND-SL  = NUMERIC DISPLAY SEPARATE SIGN LEADING
                                     ND-ST  = NUMERIC DISPLAY SEPARATE SIGN TRAILING
                                     NE     = NUMERIC EDITED
                                     NP     = NUMERIC PACKED DECIMAL UNSIGNED (COMP-3)
                  Ⓖ                  NP-S   = NUMERIC PACKED DECIMAL SIGNED
                                     *      = SUBSCRIPTED
TESTRUN  AT CARD 000071
 LOC      CARD   LV NAME                                  TYPE     VALUE

07D978   000040   02 KOUNT                                 NB-S     +01
                                                    (HEX)           0001

07D9B0   000047   02 NAME-FIELD                            AN       A

07D9EE   000053   02 NO-OF-DEPENDENTS                      AN       0

07D9B2   000049   02 RECORD-NO                             ND       0001


TESTRUN  AT CARD 000071
 LOC      CARD   LV NAME                                  TYPE     VALUE

07D978   000040   02 KOUNT                                 NB-S     +05
                                                    (HEX)           0005

07D9B0   000047   02 NAME-FIELD                            AN       E

07D9EE   000053   02 NO-OF-DEPENDENTS                      AN       4

07D9B2   000049   02 RECORD-NO                             ND       0005


TESTRUN  AT CARD 000071
 LOC      CARD   LV NAME                                  TYPE     VALUE

07D978   000040   02 KOUNT                                 NB-S     +09
                                                    (HEX)           0009

07D9B0   000047   02 NAME-FIELD                            AN       I

07D9EE   000053   02 NO-OF-DEPENDENTS                      AN       3

07D9B2   000049   02 RECORD-NO                             ND       0009
```

Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN (Part 8 of 12)

```
  IBM DOS VS COBOL                                   REL 1.0            PP NO. 5746-CB1                  07.52.05  10/02/73


TESTRUN  AT CARD 000071
 LOC      CARD    LV NAME                                  TYPE     VALUE

07D978   000040   02 KOUNT                                 NB-S     +13
                                                   (HEX)            000D

07D9B0   000047   02 NAME-FIELD                            AN       M

07D9BB   000053   02 NO-OF-DEPENDENTS                      AN       2

07D9B2   000049   02 RECORD-NO                             ND       0013


TESTRUN  AT CARD 000071
 LOC      CARD    LV NAME                                  TYPE     VALUE

07D978   000040   02 KOUNT                                 NB-S     +17
                                                   (HEX)            0011

07D9B0   000047   02 NAME-FIELD                            AN       Q

07D9BB   000053   02 NO-OF-DEPENDENTS                      AN       1

07D9B2   000049   02 RECORD-NO                             ND       0017


TESTRUN  AT CARD 000071
 LOC      CARD    LV NAME                                  TYPE     VALUE

07D978   000040   02 KOUNT                                 NB-S     +21
                                                   (HEX)            0015

07D9B0   000047   02 NAME-FIELD                            AN       U

07D9BB   000053   02 NO-OF-DEPENDENTS                      AN       0

07D9B2   000049   02 RECORD-NO                             ND       0021


TESTRUN  AT CARD 000071
 LOC      CARD    LV NAME                                  TYPE     VALUE

07D978   000040   02 KOUNT                                 NB-S     +25
                                                   (HEX)            0019

07D9B0   000047   02 NAME-FIELD                            AN       Y

07D9BB   000053   02 NO-OF-DEPENDENTS                      AN       4

07D9B2   000049   02 RECORD-NO                             ND       0025

           H
TESTRUN  AT CARD 000080
 LOC      CARD    LV NAME                                  TYPE     VALUE

         000046   01 WORK-RECORD
07D9B0                                             (HEX)            C107F0F0 F0F140D5 E8C340F0   40404040 40404040
07D9B0   000047   02 NAME-FIELD                            AN       A
07D9B1   000048   02 FILLER                                AN       *
07D9B2   000049   02 RECORD-NO                             ND       0001
07D9B6   000050   02 FILLER                                AN
07D9B7   000051   02 LOCATION                              A        NYC
07D9BA   000052   02 FILLER                                AN
07D9BB   000053   02 NO-OF-DEPENDENTS                      AN       0
07D9BD   000054   02 FILLER                                AN

07D9C8   000057   02 B                                     NP-S     *1*2*3*
                                                   (HEX)            F1F2F3C4
```

Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN (Part 9 of 12)

```
   IBM DOS VS COBOL                       REL 1.0               PP NO. 5746-CB1                07.52.05  10/02/73

                                         COBOL ABEND DIAGNOSTIC AIDS


INTERRUPT CODE 7    LAST PSW ADDR BEFORE ABEND D007E1AE          (I)

PROGRAM TESTRUN                                                           }
                                                                          } (J)
LAST CARD NUMBER/VERB NUMBER EXECUTED -- CARD NUMBER 000080/VERB NUMBER 01.}
                                                         FLOW TRACE
TESTRUN  000066 000070 000066 000070 000066 000070 000066 000070 000076 000079  (K)


                                              DATA DIVISION DUMP OF TESTRUN

     (L)
  TASK GLOBAL TABLE       LOC       VALUE

  SAVE AREA              07DC78    0000F233 1B009101 00080168 8007E18E   0007E1A8 4007E1AA 0007DA60 0007DBA8
                         07DC98    0000001A 0007DA60 5007E29A 0007D978   0007DAEC 0007DBA8 0007E264 0007D878
                         07DCB8    4007E1AE 0007DEB0
  SWITCH                 07DCC0    3C10004B
  TALLY                  07DCC4    00000000
  SORT-SAVE              07DCC8    00000000
  ENTRY-SAVE             07DCCC    0007DF38
  SORT-CORE-SIZE         07DCD0    00000000
  NSTD-REELS             07DCD4    0000
  SORT-RETURN            07DCD6    0000
  WORKING CELLS          07DCD8    0007DA60 0007DB40 0000001A 0007DA60   5007E29A F16E9025 F21290BC 40E907F0
                         07DCF8    F0F2F640 D5E8C340 F0404040 40404040   40404040 40404040 40404040 40404040
                         07DD18    40404040 40404040 40404040 40404040   40404040 40404040 40404040 40404040
                         07DD38    --SAME--
                         07DD58    40404040 40404040 40404040 40404040   40404040 40400000 0000000C 0007E082
                         07DD78    0000001A 01000000 0007E070 0007F518   0007D9AF 0007E084 0000001A 0000001A
                         07DD98    0007D9AF 5007E29A 0007D978 0007DAD8   0007DBA8 0007E264 0007D878 4007E08A
                         07DDB8    0007DEB0 55E0F088 4780F030 18FE05EF   18F498E4 F0684B00 00024620 0007E082
                         07DDD8    0007FA48 0007E084 0007F518 4A50F060   4A20F060 183047F0 F036D200 50002000
                         07DDF8    00000000 00000000 07070607 07070707
  SORT-FILE-SIZE         07DE08    00000000
  SORT-MODE-SIZE         07DE0C    00000000
  PGT-VN TBL             07DE10    E2C1D4D7
  TGT-VN TBL             07DE14    D3C54040
  SORTAB ADDR            07DE18    0007D878
  VN TBL LENGTH          07DE1C    0000
  SORTAB LENGTH          07DE1E    0000
  PROGRAM-ID             07DE20    TESTRUN
  A(INIT1)               07DE28    0007D878
  UPSI-SWITCHES          07DE2C    C9D3C2C4 D6E2E8F0
  TGT-DBG TABLE          07DE34    00000228
  CURRENT PRIORITY       07DE38    00
  TRANSIENT AREA LENGTH  07DE39    689120
  PROCEDURE-BLOCK        07DE3C    E0004780
  UNUSED                 07DE40    F0325810
  RESERVED               07DE44    F0789101
  VSAM SAVE AREA         07DE48    10004710
  UNUSED                 07DE4C    F0329601 00000000
  RESERVED               07DE54    00000000 F0549110 000002E0 00000100   000002A8 D4C5E3C8
  OVERFLOW CELLS         (NONE)
  BL CELLS               07DE6C    0007DAEC 0007DBA8 0007D978
  DTFADR CELLS           07DE78    0007D9E8 0007DA60
  FIB CELLS              (NONE)
  TEMP STORAGE           07DE80    00000000 0000026C
  BLL CELLS              07DE88    00000000
  VLC CELLS              (NONE)
  SBL CELLS              (NONE)
  INDEX CELLS            (NONE)
 OTHER (SEE MEMORY MAP)  07DE8C    0007D993 0007D9AF 0007E0AC 0007E0AC   D7C5D540 0A000A68 000009D2 5C29F0C8
                         07DEAC    000006F8
```

Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN (Part 10 of 12)

```
        IBM DOS VS COBOL                       REL 1.0            PP NO. 5746-CB1             07.52.05  10/02/73

                                                  DATA DIVISION DUMP OF TESTRUN

 LCC      CARD     LV NAME                                  TYPE    VALUE

          000017   FD FILE-1      (M)                       STANDARD SEQUENTIAL  ASSIGNED TC SYS008, CLOSED

07D9D0                                        PRE-DTF  01010014 00000000 00000000 00000000   6C000000 00000000
07D9E8                                          DTFMT  00009200 0C000108 0007DA20 0007DA28   0007E368 1160E2E8 E2F0F0F8 40400162
07DA08                                                 00000000 00000000 00000000 86BCF018   41E0E001 58201044 0107DAD8 20000064
07DA28                                                 0007DB40 0007DB40 00000014 0007DBA3   00640063 00000000 00000000 40080268
07DA48                                                 01010014 00000000 00000000 00000000   6C000000 00000000 00008200 0C000108

          000028   01 RECORD-1    (N)
07DAEC                                                 (HEX)        D807F0F0 F1F740D5 E8C340F1   40404040 40404040
07DAEC    000029   02 FIELD-A                               AN      Q*0017 NYC 1
          000018   FD FILE-2      (M)                       STANDARD SEQUENTIAL  ASSIGNED TO SYS008, OPEN INPUT

07DA48                                        PRE-DTF  01010014 00000000 00000000 00000000   6C000000 00000000
07DA60                                          DTFMT  00008200 0C000108 0007DA98 0007DAA0   0007E368 11E8E2E8 E2F0F0F8 40400272
07DA80                                                 10000000 2407E1A2 00000001 86BCF018   41E0E001 58201044 0207DC10 00000064
07DAA0                                                 0007DBA8 0007DBA8 00000014 0007DC0B   00640063 00000000 000802B0 00080268
07DAC0                                                 00000000 00000000 00000000 00000000   00000000 00000000 E907F0F0 F2F640D5

          000036   01 RECORD-2    (N)
07DBA8                                                 (HEX)        C107F0F0 F0F140D5 E8C340F0   40404040 40404040
07DBA8    000037   02 FIELD-A                               AN      A*0001 NYC 0

          000039   01 FILLER      (P)
07D978                                                 (HEX)        001AC1C2 C3C4C5C6 C7C8C9D1   D2D3D4D5 D6D7D8D9 E2E3E4E5
07D990                                                              E6E7E8E9 001AF0F1 F2F3F4F0   F1F2F3F4 F0F1F2F3 F4F0F1F2
07D9A8                                                              F3F4F0F1 F2F3F4F0
07D978    000040   02 KCUNT                                 NB-S    +26
07D97A    000041   02 ALPHABET                              AN      ABCDEFGHIJKLMNOPQRSTUVWXYZ
          000042   02 ALPHA                                *AN

                          (Q) (SUB1)
07D97A                         1                                    A
07D97B                         2                                    B
07D97C                         3                                    C
07D97D                         4                                    D
07D97E                         5                                    E
07D97F                         6                                    F
07D980                         7                                    G
07D981                         8                                    H
07D982                         9                                    I
07D983                        10                                    J
07D984                        11                                    K
07D985                        12                                    L
07D986                        13                                    M
07D987                        14                                    N
07D988                        15                                    O
07D989                        16                                    P
07D98A                        17                                    Q
```

Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN (Part 11 of 12)

```
  IBM DOS VS COBOL                          REL 1.0           PP NO. 5746-CB1              07.52.05  10/02/73

                                                  DATA DIVISION DUMP OF TESTRUN

 LCC       CARD     LV NAME                                TYPE     VALUE

07D98B                         18                                   R
07D98C                         19                                   S
07D98D                         20                                   T
07D98E                         21                                   U
07D98F                         22                                   V
07D990                         23                                   W
07D991                         24                                   X
07D992                         25                                   Y
07D993                         26                                   Z
07D994   000043   02 NUMBER                                NB-S     +26
07D996   000044   02 DEPENDENTS                            AN       01234012340123401234012340
         000045   02 DEPEND                               *AN
                      Ⓠ (SUB1)
07D996                          1                                   0
07D997                          2                                   1
07D998                          3                                   2
07D999                          4                                   3
07D99A                          5                                   4
07D99B                          6                                   0
07D99C                          7                                   1
07D99D                          8                                   2
07D99E                          9                                   3
07D99F                         10                                   4
07D9A0                         11                                   0
07D9A1                         12                                   1
07D9A2                         13                                   2
07D9A3                         14                                   3
07D9A4                         15                                   4
07D9A5                         16                                   0
07D9A6                         17                                   1
07D9A7                         18                                   2
07D9A8                         19                                   3
07D9A9                         20                                   4
07D9AA                         21                                   0
07D9AB                         22                                   1
07D9AC                         23                                   2
07D9AD                         24                                   3
07D9AE                         25                                   4
07D9AF                         26                                   0
         000046   01 WORK-RECORD Ⓟ
07D9B0                                                (HEX)         C107F0F0 F0F140D5 E8C340F0   40404040 40404040
07D9B0   000047   02 NAME-FIELD                            AN       A
07D9B1   000048   02 FILLER                                AN       *
07D9B2   000049   02 RECORD-NO                             ND       0001
07D9B6   000050   02 FILLER                                AN
07D9B7   000051   02 LOCATION                              A        NYC
07D9BA   000052   02 FILLER                                AN
07D9BE   000053   02 NO-OF-DEPENDENTS                      AN       0


                                                  DATA DIVISION DUMP OF TESTRUN

 LCC       CARD     LV NAME                                TYPE     VALUE

07D9BD   000054   02 FILLER                                AN

         000055   01 RECORDA   Ⓟ
07D9C8                                                (HEX)         F1F2F3C4
07D9C8   000056   02 A                                     ND-OT    +1234
07D9C8   000057   02 B                                 Ⓡ→ NP-S     *1*2*3*


                                                  END OF COBOL DIAGNOSTIC AIDS
```

Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN (Part 12 of 12)

A programmer using the DOS/VS COBOL Compiler and Library has several methods available to him for testing and debugging his programs. Use of the symbolic debugging features is the easiest and most efficient method for testing and debugging and is described in detail in the chapter "Symbolic Debugging Features." Using the execution statistics feature is another method for testing, debugging and optimizing a program, and is described in the chapter "Execution Statistics".

This chapter contains information useful for testing and debugging programs run without the symbolic debugging features. It also contains information on linkage editor and execution-time diagnostics as well as a description of taking checkpoints and restarting programs.

SYNTAX-CHECKING COMPILATION

The compiler checks the source text for syntax errors and then generates the appropriate error messages. With the syntax-checking feature, the programmer can request a compilation either conditionally, with object code produced only if no messages or just W- or C-level messages are generated, or unconditionally, with no object code produced regardless of message level.

Selected test cases run with the syntax-checking feature have resulted in a compilation-time saving of as much as 70%, For a discussion of the syntax-checking options, SYNTAX and CSYNTAX, see the section "CBL Statement -- COBOL Option Control Card."

IDENTIFICATION OF PROGRAM VERSIONS

One problem a programmer may have during checkout is associating a particular compilation listing with the object deck from that compilation and the output and/or dump from a particular run. To aid in this, the following facilities can be used:

1. Specify a DATE-COMPILED paragraph as part of the Environment Division. This is replaced by the actual date of compilation on the source listing (OPTION LIST).

2. The date and time of compilation are given in the header line of the compilation listing.

3. The date and time of compilation are punched into the object deck and will be found beginning at relative location X'EC' in the dump of the object module.

4. By moving the special register WHEN-COMPILED to an output record, the user may flag his output to identify it with a particular compilation. WHEN-COMPILED is described more fully in IBM DOS Full American National Standard COBOL.

DEBUG LANGUAGE

The COBOL debugging language is designed to assist the COBOL programmer in producing an error-free program in the shortest possible time. The following sections discuss the use of the debug language and other methods of program checkout.

The three debug language statements are TRACE, EXHIBIT, and ON. Any one of these statements can be used as often as necessary. They can be interspersed throughout a CCBOL source program, or they can be contained in a packet in the input stream to the compiler.

Program checkout may not be desired after testing is completed. A debug packet can be removed after testing to eliminate the extra object program coding generated for the debug statements.

The output produced by the TRACE and EXHIBIT statements is listed on the system logical output device (SYSLST).

The following discussions describe methods of using the debug language.

FLOW OF CONTROL

The READY TRACE statement causes the compiler-generated card numbers for each section-name and paragraph-name to be displayed. These card numbers are listed on SYSLST at execution time when control passes to these sections and paragraphs.

Hence, the output of the READY TRACE statement appears as a list of card numbers. If VERB is specified, the actual paragraph-names and names of the verbs will be listed.

To reduce the length of the list and the time taken to generate it, a trace can be stopped with a RESET TRACE statement. The READY TRACE/RESET TRACE combination is helpful in examining a particular area of the program where the flow of control is difficult to determine, e.g., code consists of a series of PERFORM statements or nested conditional statements. The READY TRACE statement can be coded so that the trace begins before control passes to that area. The RESET TRACE statement can be coded so that the trace stops when the program has passed beyond the area.

Use of the ON statement with the TRACE statement allows conditional control of the tracing. When the COBOL compiler encounters an ON statement, it creates a counter which is incremented during execution, whenever control passes through that ON statement. For example, if an error occurs when a specific record is processed, the ON statement can be used to isolate the problem record. The statement should be placed where control passes through it only once for each record that is read. When the contents of the counter equal the number of the record (as specified in the ON statement), a trace can be taken on that record. The following example shows a method in which the 200th record could be selected for a TRACE statement.

```
Col.
1      Area A
------------------------------------------
       RD-REC.
       .
       .
       .
DEBUG  RD-REC
       PARA-NM-1.    ON 200 READY TRACE.
                     ON 201 RESET TRACE.
```

If the TRACE statement were used without the ON statement, every record would be traced.

An example of a common program error is failing to break a loop or unintentionally creating a loop in the program. If many iterations of the loop are required before it can be determined that a program error exists, the ON statement can be used to initiate a trace after the expected number of iterations has been completed.

Note: If an error occurs during compilation of an ON statement, the diagnostic message may refer to the previous statement number.

DISPLAYING DATA VALUES DURING EXECUTION

A programmer can display the value of a data item during program execution by using the EXHIBIT statement. The EXHIBIT statement has three options:

1. EXHIBIT NAMED -- Displays the names and values of the data-names listed in the statement.
2. EXHIBIT CHANGED -- Displays the value of the data-names listed in the statement only if the value has changed since the last execution of the statement.
3. EXHIBIT CHANGED NAMED -- Displays the names and the values of the data-names only if the values have changed since the last execution of the statement.

Data values can be used to check the accuracy of the program. For example, using EXHIBIT NAMED, the programmer can display specified fields from records, compute the calculations himself, and compare his calculations with the output from his program. The coding for a payroll problem might be:

```
Col.
1      Area A
------------------------------------------

       GROSS-PAY-CALC.
           COMPUTE GROSS-PAY =
           RATE-PER-HOUR * (HRSWKD
           + 1.5 * OVERTIMEHRS).
       NET-PAY-CALC.


DEBUG  NET-PAY-CALC
       SAMPLE-1. ON 10 AND
           EVERY 10 EXHIBIT NAMED
           RATE-PER-HOUR, HRSWKD,
           OVERTIMEHRS, GROSS-PAY.
```

This coding will cause the values of the four fields to be listed for every tenth data record before net pay calculations are made. The output could appear as:

```
RATE-PER-HOUR = 4.00 HRSWKD = 40.0
     OVERTIMEHRS = 0.0 GROSS-PAY = 160.00

RATE-PER-HOUR = 4.10 HRSWKD = 40.0
     OVERTIMEHRS = 1.5 GROSS-PAY = 173.23
```

```
RATE-PER-HOUR = 3.35 HRSWKD = 40.0
     OVERTIMEHRS = 0.0 GROSS-PAY = 134.00
```

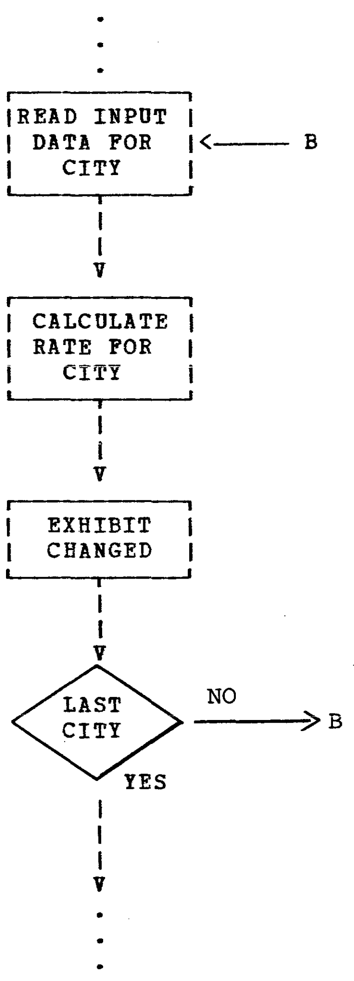
Note: Decimal points are included in this example for clarity, but actual printouts depend on the data description in the program.

The preceding was an example of checking at regular intervals (every tenth record). A check of any unusual conditions can be made by using various combinations of COBOL statements in the debug packet. For example:

```
     IF OVERTIMEHRS GREATER THAN 2.0
       EXHIBIT NAMED PAYRCDHRS...
```

In connection with the previous example, this statement could cause the entire pay record to be displayed whenever an unusual condition (overtime exceeding two hours) is encountered.

The EXHIBIT statement with the CHANGED option also can be used to monitor conditions that do not occur at regular intervals. The values of data-names are listed only if the value has changed since the last execution of the statement. For example, suppose the program calculates postage rates to various cities. The flow of the program might be:



```
    STATE = 01 CITY = 01 RAIL = 10 BUS = 14 TRUCK = 12 AIR = 20

    CITY = 02

    CITY = 03 BUS = 06 AIR = 15

    CITY = 04 RAIL = 30 BUS = 25 TRUCK = 28 AIR = 34

    STATE = 02 CITY = 01 TRUCK = 25

    CITY = 02 TRUCK = 20 AIR = 30

          .
          .
          .
```

Figure 65. Sample Output of EXHIBIT Statement With the CHANGED NAMED Option

The EXHIBIT statement with the CHANGED option in the program might be:

```
EXHIBIT CHANGED STATE CITY RATE
```

The output from the EXHIBIT statement with the CHANGED option could appear as:

```
01  01  10
    02  15
    03
    04  10
02  01
    02  20
    03  15
    04
03  01  10
     .
     .
     .
```

The first column contains the code for a state, the second column contains the code for a city, and the third column contains the code for the postage rate. The value of a data-name is listed only if it has changed since the previous execution. For example, since the postage rate to city 02 and city 03 in state 01 are the same, the rate is not printed for city 03.

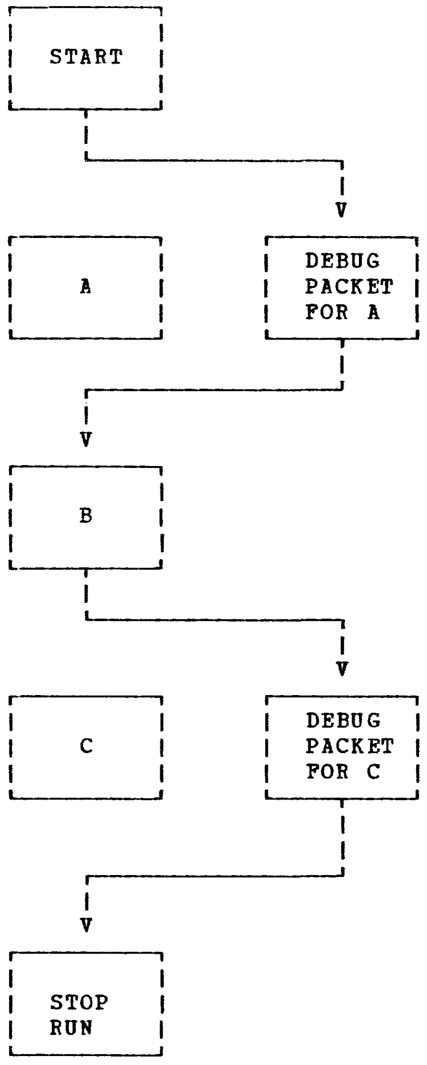
The EXHIBIT statement with the CHANGED NAMED option lists the data-name if the value has changed. For example, the program might calculate the cost of various methods of shipping to different cities. After the calculations are made, the following statement could appear in the program:

```
EXHIBIT CHANGED NAMED STATE CITY RAIL
  BUS TRUCK AIR
```

The output from this statement could appear as shown in Figure 65. Note that a data-name and its value are listed only if the value has changed since the previous execution.

TESTING A PROGRAM SELECTIVELY

A debug packet allows the programmer to select a portion of the program for testing. The packet can include test data and can specify operations the programmer wants to be performed. When the testing is completed, the packet can be removed. The flow of control can be selectively altered by the inclusion of debug packets, as illustrated in the following example of selective testing of B:



In this program, A creates data, B processes it, and C prints it. The debug packet for A simulates test data. It is first in the program to be executed. In the packet, the last statement is GO TO B, which permits A to be bypassed. After B is executed with the test data, control passes to the debug packet for C, which contains a GO TO statement that transfers control to the end of the program, bypassing C.

<u>TESTING CHANGES AND ADDITIONS TO PROGRAMS</u>

If a program runs correctly, and changes or additions might improve its efficiency, a debug packet can be used to test changes without modifying the original source program.

If the changes to be incorporated are in the middle of a paragraph, the entire paragraph with the changes included must be written in the debug packet. The last statement in the packet should be a GO TO statement that transfers control to the next procedure to be executed.

There are usually several ways to perform an operation. Alternative methods can be tested by putting them in debug packets.

The source program library facility can be used for program checkout by placing a source program in a library (see the chapter "Librarian Functions"). Changes or additions to the program can be tested by using the BASIS card and any number of INSERT and DELETE cards. Such changes or additions remain in effect only for the duration of the run.

A debug packet can also be used in conjunction with the BASIS card to debug a program or to test deletions or additions to it. The debug packet is inserted in the input stream immediately following the BASIS card and any INSERT or DELETE cards.

DUMPS

If a serious error occurs during execution of the problem program, the job is abnormally terminated; any remaining steps are bypassed; and a program phase dump is generated. The programmer can use the dump for program checkout. (However, any pending transfers to an external device may not be completed. For example, if a READY TRACE statement is in effect when the job is abnormally terminated, the last card number may not appear on the external device.) In cases where a serious error occurs in other than the problem program (for example, Supervisor), a dump is not produced. Note that program phase dumps can be suppressed if the NODUMP option of the OPTION control statement has been specified for the job, or if NODUMP was specified at system generation time and is not overridden by the DUMP option for the current job.

HOW TO USE A DUMP

When a job is abnormally terminated due to a serious error in the problem program, a message is written on SYSLST which indicates the:

1. Type of interrupt (for example, program check)
2. Hexadecimal address of the instruction that caused the interrupt
3. Condition code
4. Reason for the interrupt (for example, data exception)

The instruction address can be compared to the Procedure Division map. The contents of LISTX provide a relative address for each statement. The load address of the module (which can be obtained from the map of virtual storage generated by the Linkage Editor) must be subtracted from the instruction address to obtain the relative instruction address as shown in the Procedure Division map. The PMAP=nnnnnn CBL option can be used to relocate LISTX addresses so that this calculation need not be done. If the interrupt occurred within the COBOL program, the programmer can use the error address and LISTX to locate the specific statement in the program which caused a dump to be taken. Examination of the statement and the fields associated with it may produce information as to the specific nature of the error.

Figure 66 is a sample dump which was caused by a data exception. Invalid data (i.e., data which did not correspond to its usage) was placed in the numeric field B as a result of redefinition. The following discussion illustrates the method of finding the specific statement in the program which caused the dump. Letters identifying the text correspond to letters in the program listing.

(A) The program interrupt occurred at HEX LOCATION 07DFDC. This is indicated in the SYSLST message printed just before the dump.

(B) The linkage editor map indicates that the program was loaded into address 7D878. This is determined by examining the load point of the control section TESTRUN. TESTRUN is the name assigned to the program module by the source coding:

```
PROGRAM-ID.  TESTRUN.
```

(C) The specific instruction which caused the dump is located by subtracting the load address from the interrupt address (that is, subtracting 7D878 from 7DFDC). The result, 764, is the relative interrupt address and can be found in the object code listing. In this case the instruction in question is an AP (add decimal).

(D) The left-hand column of the object code listing gives the compiler-generated card number associated with the instruction. It is card 66. As seen in the source listing, card 66 contains the COMPUTE statement.

Additional details about reading a dump are found in the chapter "Interpreting Output."

ERRORS THAT CAN CAUSE A DUMP

A dump can be caused by one of many errors. Several of these errors may occur at the COBOL language level while others can occur at the job control level.

The following are examples of COBOL language errors that can cause a dump:

1. A GO TO statement with no procedure-name following it may have been improperly initialized with an ALTER statement. The execution of this statement will cause an invalid branch to be taken and results will be unpredictable.

2. Moves of or arithmetic calculations using numeric fields that have not been properly initialized.

   For example, neglecting to initialize the object of an OCCURS clause with the DEPENDING ON option, or referencing data fields prior to the first READ statement may cause a program interrupt and a dump.

3. Invalid data placed in a numeric field as a result of redefinition.

4. Input/output errors that are nonrecoverable.

5. Items with subscripts whose values exceed the defined maximum value can destroy machine instructions when moved.

6. Attempting to execute an invalid operation code through a system or program error.

7. Generating an invalid address for an area that has address protection.

8. Subprogram linkage declarations that are not defined exactly as they are stated in the calling program.

9. Data or instructions can be modified by entering a subprogram and manipulating data incorrectly. A COBOL subprogram can acquire invalid information from the main program, e.g., a CALL statement using a procedure-name and an ENTRY statement using a data-name.

10. An input file contains invalid data such as a blank numeric field or data incorrectly specified by its data description.

    The compiler does not generate a test to check the sign position for a valid configuration before the item is used as an operand. The programmer can test for valid data by means of the numeric class test and, by using the TRANSFORM statement, convert it to valid data under certain conditions.

    For example, if the units position of a numeric data item described as USAGE IS DISPLAY contained a blank, the blank could be transformed to a zero, thus forcing a valid sign.

11. Division by zero without an ON SIZE ERROR clause will cause a data exception.

LOCATING A DTF

One or more DTF's are generated by the compiler for each file opened in the COBOL program. All information about that file is found within the DTF or in the fields preceding the DTF. See the chapter "Detailed Processing Capabilities" for the type of information available and its location.

A particular DTF may be located in an execution-time dump as follows:

1. Determine the order of the DTF address cells in the TGT from the DTF numbers shown for each file-name in the glossary.

   Note: Since the order is the same as the FD's in the Data Division, the order can be determined from the source program if the SYM option was not used (i.e., no glossary was printed).

2. Find the relative starting address of the block of DTF cells from the TGT listing in the Memory Map.

3. Calculate the absolute starting address of the block by adding the hexadecimal relocation factor for the beginning of the object module as given in the linkage editor MAP.

4. Allowing one fullword per DTF cell, count off the cells from the starting address found in step 3, using the order determined in step 1 to locate the desired DTF cell.

5. If more than one DTF is generated for a file, the above procedure should be followed using the PGT and the SUBDTF cells rather than the TGT and the DTFADR cells. The order of multiple DTF's in storage is dependent on the OPEN option as follows:

   a. INPUT

   b. OUTPUT

   c. I-O or INPUT REVERSED

The following discussion illustrates the method of finding the DTF's in the sample program in Figure 66. Letters identifying the text refer to letters in the program listing.

Ⓔ The DTF for FILE-1 precedes the DTF for FILE-2.

Ⓕ DTFADR CELLS begin at relative location 600.

Ⓖ Since the relocation factor is 7D878, the DTFADR CELLS begin at location 7DE78 in the dump.

Ⓗ The DTF for FILE-1 begins at location 7D9E8, and the DTF for FILE-2 begins at location 7DA60.

LOCATING DATA

The location assigned to a given data-name may similarly be found by using the BL number and displacement given for that entry in the glossary, and then locating the appropriate one fullword BL cell in the TGT. The hexadecimal sum of the glossary displacement and the contents of the cell should give the relative address of the desired area. This can then be converted to an absolute address as described above.

Since the problem program in Figure 66 interrupted because of a data exception, the programmer should locate the contents of field B at the time of the interrupt. This can be done as follows:

Ⓙ Locate data-name B in the glossary. It appears under the column headed SOURCE-NAME. Source-name B has been assigned to base locator 3 (i.e., BL =3) with a displacement of 050. The sum of the value of base locator 3 and the displacement value 50 is the address of data-name B.

Ⓚ The Register Assignment table lists the registers assigned to each base locator. Register 6 has been assigned to BL =3.

Ⓛ The contents of the 16 general registers at the time of the interrupt are displayed at the beginning of the dump. Register 6 contains the address 0007D978.

Ⓜ The location of data-name B can now be determined by adding the contents of register 6 and the displacement value 50. The result, 7D9C8, is the address of the leftmost byte of the 4-byte field B.

Note: Field B contains F1F2F3C4. This is external decimal representation and does not correspond to the USAGE COMPUTATIONAL-3 defined in the source listing.

Ⓝ The location assigned to a given data-name may also be found by using the BL CELLS pointer in the TGT Memory Map. Figure 64 indicates that the BL cells begin at location 7DE6C (add 5F4 to the load point address, 7D878, of the object module). The first four bytes are the first BL cell, the second four bytes are the second BL cell, etc. Note that the third BL cell contains the value 7D978. This is the same value as that contained in register 6.

Note: Some program errors may destroy the contents of the general registers or the BL cells. In such cases, alternate methods of locating the DTF's are useful.

```
// JOB DATACHK
// OPTION NODECK,LINK,LIST,LISTX,SYM,ERRS
// EXEC FCOBOL




    1  IBM DOS VS COBOL                          REL 1.0          PP NO. 5746-CB1                    07.35.08  10/02/73

 CBL QUOTE,SEQ
00001          IDENTIFICATION DIVISION.
00002          PROGRAM-ID.  TESTRUN.
00003              AUTHOR.  PROGRAMMER NAME.
00004              INSTALLATION.  NEW YORK PROGRAMMING CENTER.
00005              DATE-WRITTEN.  SEPTEMBER 25,1973
00006          DATE-COMPILED. 10/02/73
00010          ENVIRONMENT DIVISION.
00011          CONFIGURATION SECTION.
00012          SOURCE-COMPUTER.  IBM-370.
00013          OBJECT-COMPUTER.  IBM-370.
00014          INPUT-OUTPUT SECTION.
00015          FILE-CONTROL.
00016              SELECT FILE-1 ASSIGN TO SYS008-UT-2400-S.
00017              SELECT FILE-2 ASSIGN TO SYS008-UT-2400-S.
00018          DATA DIVISION.                            \
00019          FILE SECTION.                              |
00020          FD  FILE-1                                 |
00021              LABEL RECORDS ARE OMITTED              |
00022              BLOCK CONTAINS 5 RECORDS               |
00023              RECORDING MODE IS F                    |
00024              RECORD CONTAINS 20 CHARACTERS          |
00025              DATA RECORD IS RECORD-1.               |
00026          01  RECORD-1.                              |
00027              05 FIELD-A PIC X(20).                   >  (E)
00028          FD  FILE-2                                 |
00029              LABEL RECORDS ARE OMITTED              |
00030              BLOCK CONTAINS 5 RECORDS               |
00031              RECORD CONTAINS 20 CHARACTERS          |
00032              RECORDING MODE IS F                    |
00033              DATA RECORD IS RECORD-2.               |
00034          01  RECORD-2.                              |
00035              05 FIELD-A PIC X(20).                 /
```

Figure 66. Sample Dump Resulting from Abnormal Termination (Part 1 of 6)

```
IBM DOS VS COBOL                                    REL 1.0         PP NO. 5746-CB1                 07.35.08  10/02/73

C0036          WORKING-STORAGE SECTION.
00037          01  FILLER.
00038              02 KOUNT PIC S99 COMP SYNC.
C0039              02 ALPHABET PIC X(26) VALUE IS "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
C0040              02 ALPHA REDEFINES ALPHABET PIC X OCCURS 26 TIMES.
C0041              02 NUMBR PIC S99 COMP SYNC.
C0042              02 DEPENDENTS PIC X(26) VALUE "01234012340123401234012340".
C0043              02 DEPEND REDEFINES DEPENDENTS PIC X OCCURS 26 TIMES.
C0044          01  WORK-RECORD.
00045              05 NAME-FIELD PIC X.
C0046              05 FILLER PIC X.
00047              05 RECORD-NO PIC 9999.
00048              05 FILLER PIC X VALUE IS SPACE.
00049              05 LOCATION PIC AAA VALUE IS "NYC".
C0050              05 FILLER PIC X VALUE IS SPACE.
C0051              05 NO-OF-DEPENDENTS PIC XX.
00052              05 FILLER PIC X(7) VALUE IS SPACES.
00053          01  RECORDA.
C0054              02 A PICTURE S9(4) VALUE 1234.
C0055              02 B REDEFINES A PICTURE S9(7) COMPUTATIONAL-3.
C0056          PROCEDURE DIVISION.
C0057          BEGIN. READY TRACE.
C0058              NOTE THAT THE FOLLOWING OPENS THE OUTPUT FILE TO BE CREATED
C0059              AND INITIALIZES THE COUNTERS.
C0060          STEP-1. OPEN OUTPUT FILE-1. MOVE ZERO TO KOUNT, NUMBR.
C0061              NOTE THAT THE FOLLOWING CREATES INTERNALLY THE RECORDS TO BE
C0062              CONTAINED IN THE FILE, WRITES THEM ON TAPE, AND DISPLAYS
C0063              THEM ON THE CONSOLE.
C0064          STEP-2. ADD 1 TO KOUNT, NUMBR. MOVE ALPHA (KOUNT) TO
C0065              NAME-FIELD.
C0066                  COMPUTE B = B + 1.  <-----(D)
C0067              MOVE DEPEND (KOUNT) TO NO-OF-DEPENDENTS.
C0068              MOVE NUMBR TO RECORD-NO.
C0069          STEP-3. DISPLAY WORK-RECORD UPON CONSOLE. WRITE RECORD-1 FROM
C0070              WORK-RECORD.
C0071          STEP-4. PERFORM STEP-2 THRU STEP-3 UNTIL KOUNT IS EQUAL TO 26.
C0072              NOTE THAT THE FOLLOWING CLOSES THE OUTPUT FILE AND REOPENS
C0073              IT AS INPUT.
C0074          STEP-5. CLOSE FILE-1. OPEN INPUT FILE-2.
C0075              NOTE THAT THE FOLLOWING READS BACK THE FILE AND SINGLES
00076              OUT EMPLOYEES WITH NO DEPENDENTS.
00077          STEP-6. READ FILE-2 RECORD INTO WORK-RECORD AT END GO TO STEP-8.
C0078          STEP-7. IF NO-OF-DEPENDENTS IS EQUAL TO "0" MOVE "Z" TO
00079              NO-OF-DEPENDENTS. EXHIBIT NAMED WORK-RECORD. GO TO STEP-6.
C0080          STEP-8. CLOSE FILE-2.
C0081              STOP RUN.
```

Figure 66. Sample Dump Resulting from Abnormal Termination (Part 2 of 6)

```
 IBM DOS VS COBOL                      REL 1.0            PP NO. 5746-CB1            07.35.08  10/02/73

INTRNL NAME    LVL SOURCE NAME                     BASE    DISPL   INTRNL NAME    DEFINITION    USAGE       R  O  Q  M


DNM=1-148      FD  FILE-1                          DTF=01          DNM=1-148                    DTFMT                F
DNM=1-179      01  RECORD-1                         BL=1    000    DNM=1-179      DS 0CL20      GROUP
DNM=1-200      02  FIELD-A                          BL=1    000    DNM=1-200      DS 20C        DISP
DNM=1-217      FD  FILE-2                          DTF=02          DNM=1-217                    DTFMT                F
DNM=1-248      01  RECORD-2                         BL=2    000    DNM=1-248      DS 0CL20      GROUP
DNM=1-269      02  FIELD-A                          BL=2    000    DNM=1-269      DS 20C        DISP
DNM=1-289      01  FILLER                           BL=3    000    DNM=1-289      DS 0CL56      GROUP
DNM=1-308      02  KOUNT                            BL=3    000    DNM=1-308      DS 1H         COMP
DNM=1-323      02  ALPHABET                         BL=3    002    DNM=1-323      DS 26C        DISP
DNM=1-341      02  ALPHA                            BL=3    002    DNM=1-341      DS 1C         DISP        R  O
DNM=1-359      02  NUMBR                            BL=3    01C    DNM=1-359      DS 1H         COMP
DNM=1-374      02  DEPENDENTS                       BL=3    01E    DNM=1-374      DS 26C        DISP
DNM=1-394      02  DEPEND                           BL=3    01E    DNM=1-394      DS 1C         DISP        R  O
DNM=1-410      01  WORK-RECORD                      BL=3    038    DNM=1-410      DS 0CL20      GROUP
DNM=1-434      02  NAME-FIELD                       BL=3    038    DNM=1-434      DS 1C         DISP
DNM=1-454      02  FILLER                           BL=3    039    DNM=1-454      DS 1C         DISP
DNM=1-473      02  RECORD-NO                        BL=3    03A    DNM=1-473      DS 4C         DISP-NM
DNM=1-492      02  FILLER                           BL=3    03E    DNM=1-492      DS 1C         DISP
DNM=2-000      02  LOCATION                         BL=3    03F    DNM=2-000      DS 3C         DISP
DNM=2-018      02  FILLER                           BL=3    042    DNM=2-018      DS 1C         DISP
DNM=2-037      02  NO-OF-DEPENDENTS                 BL=3    043    DNM=2-037      DS 2C         DISP
DNM=2-063      02  FILLER                           BL=3    045    DNM=2-063      DS 7C         DISP
DNM=2-082      01  RECORDA                          BL=3    050    DNM=2-082      DS 0CL4       GROUP
DNM=2-102      02  A                                BL=3    050    DNM=2-102      DS 4C         DISP-NM
DNM=2-113      02  B  <------(J)                    BL=3    050    DNM=2-113      DS 4P         COMP-3      R


                MEMORY MAP

             TGT                     00400

          SAVE AREA                  00400
          SWITCH                     00448
          TALLY                      0044C
          SORT SAVE                  00450
          ENTRY-SAVE                 00454
          SORT CORE SIZE             00458
          NSTD-REELS                 0045C
          SORT RET                   0045E
          WORKING CELLS              00460
          SORT FILE SIZE             00590
          SORT MODE SIZE             00594
          PGT-VN TBL                 00598
          TGT-VN TBL                 0059C
          SORTAB ADDRESS             005A0
          LENGTH OF VN TBL           005A4
          LNGTH OF SORTAB            005A6
          PGM ID                     005A8
          A(INIT1)                   005B0
          UPSI SWITCHES              005B4
          DEBUG TABLE PTR            005BC
          CURRENT PRIORITY           005C0
          TA LENGTH                  005C1
          PROCEDURE BLOCK1 PTR       005C4
          UNUSED                     005C8
          RESERVED                   005CC
          VSAM SAVE AREA ADDRESS     005D0
          UNUSED                     005D4
          RESERVED                   005DC
          OVERFLOW CELLS             005F4
          BL CELLS  <----(N)         005F4
          DTFADR CELLS               00600  <-----(F)
          FIB CELLS                  00608
          TEMP STORAGE               00608
          TEMP STORAGE-2             00610
          TEMP STORAGE-3             00610
          TEMP STORAGE-4             00610
          BLL CELLS                  00610
          VLC CELLS                  00614
          SBL CELLS                  00614
          INDEX CELLS                00614
          SUBADR CELLS               00614
          ONCTL CELLS                0061C
          PFMCTL CELLS               0061C
          PFMSAV CELLS               0061C
          VN CELLS                   00620
          SAVE AREA =2               00624
          XSASW CELLS                00624
          XSA CELLS                  00624
          PARAM CELLS                00624
          RPTSAV AREA                00628
          CHECKPT CTR                00628
          IOPTR CELLS                00628
```

Figure 66. Sample Dump Resulting from Abnormal Termination (Part 3 of 6)

```
 REGISTER ASSIGNMENT

  REG 6    BL =3  <----(K)
  REG 7    BL =1
  REG 8    BL =2


64
               000708  48 30 C 03A                 LH     3,03A(0,12)         LIT+2
               00070C  4A 30 6 000                 AH     3,000(0,6)          DNM=1-308
               000710  4E 30 D 208                 CVD    3,208(0,13)         TS=01
               000714  D7 05 D 208 D 208           XC     208(6,13),208(13)   TS=01         TS=01
               00071A  94 0F D 20E                 NI     20E(13),X'0F'       TS=01+6
               00071E  4F 30 D 208                 CVB    3,208(0,13)         TS=01
               000722  40 30 6 000                 STH    3,000(0,6)          DNM=1-308
               000726  48 30 C 03A                 LH     3,03A(0,12)         LIT+2
               00072A  4A 30 6 01C                 AH     3,01C(0,6)          DNM=1-359
               00072E  4E 30 D 208                 CVD    3,208(0,13)         TS=01
               000732  D7 05 D 208 D 208           XC     208(6,13),208(13)   TS=01         TS=01
               000738  94 0F D 20E                 NI     20E(13),X'0F'       TS=01+6
               00073C  4F 30 D 208                 CVB    3,208(0,13)         TS=01
               000740  40 30 6 01C                 STH    3,01C(0,6)          DNM=1-359
64
               000744  41 40 6 002                 LA     4,002(0,6)          DNM=1-341
               000748  48 20 6 000                 LH     2,000(0,6)          DNM=1-308
               00074C  4C 20 C 03A                 MH     2,03A(0,12)         LIT+2
               000750  1A 42                       AR     4,2
               000752  5B 40 C 038                 S      4,038(0,12)         LIT+0
               000756  50 40 D 214                 ST     4,214(0,13)         SBS=1
               00075A  58 E0 D 214                 L      14,214(0,13)        SBS=1
               00075E  D2 00 6 038 E 000           MVC    038(1,6),000(14)    DNM=1-434     DNM=1-341
66
               000764  FA 30 6 050 C 03C   (C)---> AP     050(4,6),03C(1,12)  DNM=2-113     LIT+4
67


// EXEC LNKEDT


         PHASE  XFR-AD  LOCORE  HICORE  DSK-AD    ESD TYPE  LABEL     LOADED  REL-FR

       PHASE***  07D878  07D878  07F2AF  05F 0F 4  CSECT     TESTRUN   07D878  07D878  RELOCATABLE    <----(B)

                                                  CSECT     IJFFBZZN  07E278  07E278
                                                  * ENTRY   IJFFZZZN  07E278
                                                  * ENTRY   IJFFBZZZ  07E278
                                                  * ENTRY   IJFFZZZZ  07E278

                                                  CSECT     ILBDSAE0  07F128  07F128
                                                    ENTRY   ILBDSAE1  07F170

                                                  CSECT     ILBDMNS0  07F120  07F120

                                                  CSECT     ILBDDSP0  07E628  07E628
                                                    ENTRY   ILBDDSP1  07EA28

                                                  CSECT     ILBDIML0  07F0C8  07F0C8

                                                  CSECT     ILBDDSS0  07EDA0  07EDA0
                                                    ENTRY   ILBDDSS1  07F000
                                                    ENTRY   ILBDDSS2  07EFF8
                                                    ENTRY   ILBDDSS3  07F0B8
                                                    ENTRY   ILBDDSS4  07EDC6
                                                    ENTRY   ILBDDSS5  07EE72
                                                    ENTRY   ILBDDSS6  07EED2
                                                    ENTRY   ILBDDSS7  07EE9C
                                                    ENTRY   ILBDDSS8  07EDF6

                                                  CSECT     IJJCPDV   07EB58  07EB58
                                                    ENTRY   IJJCPDV1  07EB58
                                                  * ENTRY   IJJCPDV2  07EB58
* UNREFERENCED SYMBOLS                            WXTRN     STXITPSW
                                                  WXTRN     ILBDDBG2

002 UNRESOLVED ADDRESS CONSTANTS


// ASSGN SYS008,X'482'
// EXEC
```

Figure 66. Sample Dump Resulting from Abnormal Termination (Part 4 of 6)

```
0S03I PROGRAM CHECK INTERRUPTION - HEX LOCATION 07DFDC - CONDITION CODE 2 - DATA EXCEPTION  Ⓐ
0S00I JOB DATACHK  CANCELED


        DATACHK                                                                    Ⓛ

GR 0-F   0007DE78 0007DF80 00000001 00000001   0007D97A 5007E22C 0007D978 0007DB40
         0007DBA8 0007E1FC 0007D878 0007D878   0007DEA0 0007DC78 0007D97A 0007E628
FP REG   00000000 00000000 00000000 00000000   00000000 00000000 00000000 00000000
CR 0-F   004000FF 0100DD00 FFFFFFFF FFFFFFFF   00000000 00000000 00000000 00000000
         00000000 00000000 00000000 00000000   00000000 00000000 C4000000 00000200
COMREG   BG ADDR IS 000360


000000   00000000 00000000 00000000 00000000   00000000 00000360 00000000 00000000
000020   070D0000 000076CE 040D0000 000074DA   00000000 00000000 070C2000 0000071E
000040   0000DE68 08000000 0000DE58 00000000   EFFCC798 012A88EA 040C0000 000009BE
000060   040C0000 0000097A 000C0000 0000980C   04080000 0000C09E 040C0000 000008F8
000080   00000400 00000000 00020007 00020001   1207F003 00000000 00000000 00000000
0000A0   00000000 00000000 00000000 000002C0   00000000 00000000 0000000E 00000000
0000C0   00000000 --SAME--
000360   F1F061F0 F261F7F3 F000F000 00000000   00000000 00000000 C4C1E3C1 C3C8D240
000380   0011D7FF 0007F2AF 0007F2AF 00000010   0017D7FF F97F5CD3 A8A07CD0 00A63891
0003A0   38983D4A 3D4B0000 374C3750 375438F1   F0F0F2F7 F3F2F7F5 00003544 00000000
0003C0   3EF00000 363C36BC 372C0010 00000010   000070EC 00000000 000030DC 34440000
0003E0   00000000 036010E0 00000448 C0C00340   40404040 40404000 40404040 40404000
000400   0000528C 00002742 063E05FA 00003A40   00001F42 00004058 00004F5E 000073D8
000420   0000F000 0300505E 0014002C 00030000   00005F78 00000000 000072F5 00005204
000440   40800000 0000070E 00008350 00005100   000051B4 0000520C 00100010 00000000
000460   00000000 000065F8 00000000 000030D0   000004F0 000070F5 00003C5A 00000000
000480   00009812 000088C8 00000544 00000000   0000BC88 00007F50 000032FC 00003314
0004A0   00081018 00200000 00000000 00000000   00000000 000031A6 00000000 00000000
0004C0   00000000 00000000 00000005 03C10401   0000DD00 0000CDF0 0000D5F0 00005378
0004E0   000053B8 00000000 00000000 0007D7FF   FF010174 015502A0 FF000045 0000C6E0
000500   0000C0A0 00003868 0000A1E8 00000000   000A0000 00001000 00002000 00003000
000520   070D0000 000076C4 00000360 00000000   8000750E 900075C0 400053C8 00004F5E
000540   00080700 00000000 00000000 00000000   04FC0552 00000000 00000000 00000000
000560   0000318C 0000DEE8 00000008 00020406   080A0C0E 00183048 607890A8 00000000
000580   00000000 00830083 00830083 00830083   00830083 00830083 00830083 00830083
0005A0   00830083 --SAME--
0005C0   00830083 00830083 00000083 83838300   80020000 00007888 00000000 0000BBA5
0005E0   06B006B0 06B006B0 06B006B0 4EB0056A   06B006B0 06B006B0 06B006B0 06B006B0
000600   06B006B0 06B006B0 06B006B0 06B006B0   41BB001F 41BB0010 18F69503 04454770
000620   06604590 06D447F0 066006B0 06B006B0   06B006B0 06B006B0 06B041BB 001F06B0
000640   06B041BB 001B4570 06A85890 041441F0   92161211 47700660 41F09314 94633006
000660   42B0A001 58B00514 960CA00F 07FF960C   A00F4400 B528077F 920003CF 928305C9
```

Figure 66. Sample Dump Resulting from Abnormal Termination (Part 5 of 6)

```
        DATACHK

  07D800   D7C8C1E2 C55C5C5C 071D2000 0007DFE2    0007E1FC 0007D878 0007D878 0007DEA0
  07D820   0007DC78 0007D97A 0007E628 0007DE78    0007DF80 00000001 00000001 0007D97A
  07D840   5007E22C 0007D978 0007DB40 0007DBA8    0000844B E56753BE 00000000 00000000
  07D860   00000000 00000000 00000000 00000000    00000000 00000000 05F00700 900EF00A
  07D880   47F0F082 0007D878 0007D878 0011D7FF    0000D8E6 0000DB64 80000015 80000015
  07D8A0   0011D7FF 0007F8C8 0A16180C 00000000    182F07F1 0007D878 D7C8C1E2 C55C5C5C
  07D8C0   00000000 00000000 00000000 00000000    00000000 00000000 0007D978 0007DAD8
  07D8E0   0007DBA8 0007E1FC 0007D878 0007D878    0007DEA0 0007DC78 00000000 58C0F0C6
  07D900   58E0C000 58D0F0CA 9500E000 4770F0A2    9610D048 92FFE000 47F0F0AC 98CEF03A
  07D920   90ECD00C 185D989F F0BA9110 D0480719    07FF0700 0007E1FC 0007D878 0007D878
  07D940   0007DEA0 0007DC78 0007DF08 0007E1E2    C3D6C2C6 F3F0F0F0 E3C5E2E3 D9E4D540
  07D960   00000000 F1F061F0 F261F7F3 F0F74BF3    F54BF0F8 10604780 0001C1C2 C3C4C5C6
  07D980   C7C8C9D1 D2D3D4D5 D6D7D8D9 E2E3E4E5    E6E7E8E9 0001F0F1 F2F3F4F0 F1F2F3F4
  07D9A0   F0F1F2F3 F4F0F1F2 F3F4F0F1 F2F3F4F0 M  C107DB78 000040D5 E8C34000 00404040
I 07D9C0   40404040 00000000 F1F2F3C4 00000000    01010014 00000000 00000000 00000000
  07D9E0   6C000000 00000000 00009200 00000108    0007DA20 00000000 0007E278 1160E2E8
  07DA00   E2F0F0F8 40400162 10000000 04000000    00000000 86BCF018 41E0E001 58201044
  07DA20   0107DAD8 20000064 0007DB40 0007DB40    00000014 0007DBA3 00640063 00000000



  07DA40   00000000 4007F128 01010014 00000000    00000000 00000000 00000000 00000000
  07DA60   00008200 00000108 0007DA98 00000000    0007E278 1168E2E8 E2F0F0F8 40400272
  07DA80   00000000 20000000 00000000 86BCF018    41E0E001 58201044 0207DBA8 00000064
  07DAA0   0007DC10 00000000 00000014 00000000    00640063 00000000 0007F170 0007F128
  07DAC0   00000000 --SAME--
  07DB00   00000000 00000000 00008000 00000107    0007DB90 00000000 0007E450 02050202
  07DB20   0007DCF8 00000000 0207DD10 20000014    4120E000 47000000 0000FFFF FFFFFFFF
  07DB40   FFFFFFFF FFFFFFFF FFFFDBE1 20000082    4710F132 47F0F15E 91084015 00000000
  07DB60   00008000 00000107 0007DBE8 00000000    0007E3B8 02050202 0007DCE0 00000000
  07DB80   0207DD38 20000050 4120E000 47000000    0000F0C4 18E44BE0 F24C430E 00008900
  07DBA0   00198800 00190600 12004780 F074420E    00009601 F23247F0 F0464800 F23047F0
  07DBC0   F14C9108 40154710 F0949180 40154710    F0E84100 000847F0 F0C418E4 4BE0F24E
  07DBE0   430E0001 89000019 88000019 43EE0000    89E00019 88E00019 19E04780 F08C4100
  07DC00   001447F0 F0C44100 000C1B40 58E40000    1A4041EE 000012EE 4780F132 50E0F1F2
  07DC20   58140038 5010F20A 4110F20A 5010F202    D201F200 40069108 40154780 F11C18E4
  07DC40   4BE0F250 D201F210 E0005810 F2420A00    91801002 4710F114 0A079108 40204710
  07DC60   F1A85820 F20A4122 000058E0 F23E58F0    F1F207FF 47F0F104 0000F233 1B009101
  07DC80   F2324780 F1444800 F22E9400 F2325820    F23A58E0 F23E58F4 00100A09 41E0F11C
  07DCA0   47F0F16E 41E0F14C 9101F233 4780F16E    0010004B 00000000 00000000 00000000
  07DCC0   7010004B 00000000 00000000 0007DF08    00000000 00000000 0007D9E8 0007DB40
  07DCE0   0000D8E6 0007D9E8 5007E22C F16E9025    F21290BC 40404040 40404040 40404040
  07DD00   40404040 --SAME--
  07DD60   40404040 40404040 40404040 40400000    00000000 0007DF78 0007DB40 01000000
  07DD80   4007DF54 0007E628 0007DE78 0007DF80    0007DB40 0000D8E6 0007D9E8 5007E22C
  07DDA0   0007D978 0007DB40 0007DBA8 0007E1FC    0007D878 0007D878 0007DEA0 0007DF80
  07DDC0   0007E628 0007DE78 0007F000 0007DB40    00014720 0007DF78 0007EDA0 4007E88E
  07DDE0   50002000 4A50F060 4A20F060 183047F0    F036D200 50002000 00000000 00000000
  07DE00   07070607 07070707 00000000 00000000    E2C1D4D7 D3C54040 0007D878 00000000
  07DE20   E3C5E2E3 D9E4D540 0007D878 C9D3C2C4    D6E2E8F0 F3F09002 D0689120 E0004780
  07DE40   F0325810 F0789101 10004710 F0329601    00000000 00000000 F0549110 000002E0
N 07DE60   00000100 000002A8 D4C5E3C8 0007DB40    0007DBA8 0007D978 0007D9E8 0007DA60  G
  07DE80   00000000 0000001C 00000000 0007D97A    9802D068 47FE0002 0007E05E D7C5D540
  07DEA0   0007F120 0007E628 0007F0C8 0007DF70    0007E05E 0007E100 0007E14C 0007E1A4
  07DEC0   0007E07E 0007E092 0007E146 0007E174    0007E05E 400B1F88 00000001 1C00001A
  07DEE0   5B5BC2D6 D7C5D540 5B5BC2C3 D3D6E2C5    5B5BC2C6 C3D4E4D3 C0000000 E6D6D9D2
  07DF00   60D9C5C3 D6D9C4BE 58F0C004 051F0001    4004F5F7 404040D2 9640D048 58F0C004
  07DF20   051F0001 4004F6F0 40404076 5820D1F4    4110C040 5800D200 184005F0 5000F008
  07DF40   4500F00C 0007D9E8 0A024100 D20058F0    C00805EF 5810D200 96101020 5020D1F4
  07DF60   5870D1F4 D2016000 C038D201 601CC038    58F0C004 051F0001 4004F6F4 40404004
  07DF80   4830C03A 4A306000 4E30D208 D705D208    D208940F D20E4F30 D2084030 60004830
  07DFA0   C03A4A30 601C4E30 D208D705 D208D208    940FD20E 4F30D208 4030601C 41406002
  07DFC0   48206000 4C20C03A 1A425B40 C0385040    D21458E0 D214D200 6038E000 FA306050
  07DFE0   C03C4140 601E4820 60004C20 C03A1A42    5B40C038 5040D218 58E0D218 D2006043
  07E000   E0009240 60444830 601C4E30 D208F331    603AD20E 96F0603D 58F0C004 051F0001
  07E020   4004F6F9 404040F0 58F0C004 051F0002    00000014 0D0001FC 0038FFFF D2137000
  07E040   60385810 D2001841 58F01010 45E0F00C    5020D1F4 5870D1F4 5810D220 07F158F0
  07E060   C004051F 00014004 F7F14040 400F5800    D2205000 D21C5800 C0205000 D2204830
  07E080   60004930 C03E58F0 C024078F 5810C00C    07F15800 D21C5000 D22058F0 C004051F
  07E0A0   00014004 F7F44040 40585810 D20094EF    10201801 18404110 C04805F0 5000F008
  07E0C0   4500F00C 00000000 0A025800 D2004110    C0500A02 4110C040 5800D204 184005F0
  07E0E0   5000F008 4500F00C 00000000 0A024100    D20458F0 C00805EF 5810D204 96101020
  07E100   58F0C004 051F0001 4004F7F7 40404040    5810D204 58F0C028 91201010 071F1841
  07E120   41F0C028 D2021025 F00158F0 101045E0    F0085020 D1F85880 D1F8D213 60388000
  07E140   58F0C018 07FF5810 C01C07F1 58F0C004    051F0001 4004F7F8 40404080 5820C02C
```

Figure 66. Sample Dump Resulting from Abnormal Termination (Part 6 of 6)

EXECUTION STATISTICS

The DOS/VS COBOL Compiler provides several methods for testing, debugging, and optimizing programs. Use of the symbolic debugging features is an efficient method for testing and debugging a program, and is described in the chapter "Symbolic Debugging Features". The chapter entitled "Program Checkout" contains information useful for testing and debugging programs without the symbolic debugging features. The OPT option, described in the chapter "Preparing COBOL Programs for Processing", is an efficient method for automatically optimizing a program.

This chapter describes execution statistics -- how they may be obtained, some sample output, and some uses of the output.

OBTAINING EXECUTION STATISTICS

Execution statistics are invoked via the CBL card at compile time. No source language coding changes are required. The execution frequency statistics option, COUNT, facilitates testing, debugging, and optimizing by providing the programmer with verb counts at the following times.

- STOP RUN
- GOBACK in the main program
- Abnormal termination of a job

When COUNT is specified, the following items should be taken into account:

1. If COUNT and STXIT are desired, either STIXIT must be requested in the program unit requesting COUNT, or, the program unit requesting COUNT must be entered before the program unit requesting STIXIT.
2. When COUNT is specified, the compiler divides the program into blocks of verbs. When the statistics are printed, the last block of verbs executed in each program unit is indicated. If the program abnormally terminates, the statement causing the abnormal termination can be determined (by using the symbolic debugging features, for example). The programmer should then subtract one from the verb count for each verb flagged which follows the abending verb.
3. To obtain execution statistics if COUNT is requested for one of many program units, either all programs must be compiled by at least DOS/VS Release 2 compiler, or the program must terminate in a program unit compiled on at least a DOS/VS COBOL Release 2 compiler, or the program must terminate in at least a DOS/VS COBOL library Release 2 subroutine.
4. If COUNT is requested, the user must specify the SIZE parameter on his load module EXEC card. The dynamic space required for COUNT is approximately 512 bytes plus 80 bytes per program unit being monitored, and four bytes per count block (see the compiler output statistics). The requirements for each program unit are rounded to the next 128-byte boundary.
5. The OTHERWISE verb is treated as if the user coded the ELSE verb.

Debugging and Testing

The execution statistics clearly identify the following areas of the program:

- Untested and weakly tested areas of the program
- The last blocks entered and executed
- Possible sources of unnecessary code
- The most heavily used parts of the program; that is, those parts most susceptible to changes.

OPTIMIZATION METHODS

Based on execution frequency and timer statistics, the following types of optimization can be implemented by the user:

- Resequencing the program
- Insight into SYMDMP
- Common expression elimination
- Backward movement
- Unrolling
- Jamming
- Unswitching
- Incorporating procedures inline
- Tabling
- Efficiency guidelines

Note, however, that each optimization technique can result in more inefficient code if the statistics used in optimizing the program are not representative of the normal program flow. In addition, it is recommended that any optimization methods implemented be documented in the program.

Resequencing the Program

The COBOL Procedure Division should be organized as follows:

1. All frequently-used paragraphs or sections should be located near the routines that use them.
2. All infrequently-used paragraphs or sections should be grouped together and apart from frequently-used routines.
3. The most frequently-referenced data items should be placed in the beginning of the Working-Storage Sections.

Insight into SYMDMP Output

The area where dynamic symbolic dumps are to be used can be pointed to by the execution statistics. Knowledge of what area of code is executed and how often it is executed should give the user information on what sections should be further investigated.

Common Expression Elimination

This technique is designed to eliminate unnecessary arithmetic calculations. An arithmetic expression calculation is considered unnecessary if it represents a value calculated elsewhere that will always be used without modification. One such example would be an arithmetic expression whose operands are not redefined or reevaluated, but the expression is recalculated.

Backward Movement

This technique facilitates moving calculations and other operations from an area of code frequently executed to an area less frequently executed. For example, an expression calculated within a PERFORMed procedure (using a Format 2, 3, or 4 PERFORM statement) which always yields the same value for that PERFORM statement could be calculated in-line or in another procedure which would be PERFORMed just prior to the regularly PERFORMed procedure. Another example might be an expression which is calculated in many procedures which are often PERFORMed in succession. This expression could be removed from all the procedures and calculated just once prior to the procedures.

Unrolling

Procedures which are frequently executed may be expanded so that the statements within the procedure are repeated, with slight modification, to reduce the procedure overhead. For example,

```
    PERFORM YEARLY-GROSS-CALC VARYING
            WEEK-NO
            FROM 1 BY 1 UNTIL WEEK-NO
            GREATER THAN 52.

    YEARLY-GROSS-CALC.
            ADD GROSS-SALARY (WEEK-NO) TO
            YEARLY-GROSS
```

could be replaced by

```
    PERFORM YEARLY-GROSS-CALC VARYING
            WEEK-NO
            FROM 1 BY 4 UNTIL WEEK-NO
            GREATER THAN 52.

    YEARLY-GROSS-CALC.

    ADD     GROSS-SALARY (WEEK-NO),
            GROSS-SALARY (WEEK-NO+1),
          GROSS-SALARY (WEEK-NO+2), GROSS
          SALARY (WEEK-NO+3)
          YEARLY-GROSS.
```

In addition, indexing might be useful in this example.

Jamming

In some instances, two procedures can be merged into one procedure, thereby saving some procedure overhead. An example of this might be replacing

```
    MOVE 0 TO WEEK-NUM.
    PERFORM YEARLY-GROSS-CAL 52 TIMES.
    MOVE 0 TO WEEK-NUM.
    PERFORM YEARLY-NET-CAL 52 TIMES.
            .
            .
            .
    YEARLY-GROSS-CAL.
       ADD 1 TO WEEK-NUM.
       ADD GROSS-SALARY (WEEK-NUM) to
       YEARLY-GROSS.
    YEARLY-NET-CAL.
       ADD 1 TO WEEK-NUM.
       ADD NET-SALARY (WEEK-NUM) TO
       YEARLY-NET.
```

by

```
    MOVE 0 TO WEEK-NUM.
    PERFORM YEARLY-CAL 52 TIMES.
            .
            .
            .
    YEARLY-CAL.
       ADD 1 TO WEEK-NUM.
       ADD GROSS-SALARY (WEEK-NUM) to
       YEARLY-GROSS.
       ADD NET-SALARY (WEEK-NUM) TO
       YEARLY-NET.
```

Unswitching

Procedures may contain tests that result in the same action for any set of executions of that procedure. In such a case, the test can be removed from the procedure and the procedure duplicated. For example, if "SWITCH" is not changed within the loop, replace

```
       COUNT=0
       PERFORM  JOBS-TOTAL-CAL JOB-NUM
       TIMES.
            .
            .
            .
    JOB-TOTAL-CAL.
       ADD 1 TO COUNT.
       ADD JOB-COST (COUNT) TO
       TOTAL-JOB-COST.
       IF SWITCH = 0 ADD JOB-EXPENSE
       (COUNT) TO TOTAL-EXPENSES ELSE
       ADD JOB-EXPENSE (COUNT) OVERHEAD TO
       TOTAL-EXPENSES.
       ADD JOB-INCOME (COUNT) TO
       TOTAL-INCOME.
       IF SWITCH = 0 ADD JOB-PROFIT (COUNT)
       TO TOTAL-PROFITS ELSE
       COMPUTE TOTAL-PROFITS =
       TOTAL-PROFITS + JOB-INCOME (COUNT)
       - JOB-COST (COUNT) - JOB-EXPENSE
       (COUNT) - OVERHEAD.
```

by

```
       COUNT = 0
       IF SWITCH = 0
          PERFORM JOB-TOTAL-CAL-0 JOB-NUM
          TIMES ELSE
          PERFORM JOB-TOTAL-CAL-1 JOB-NUM
          TIMES.
          .
          .
          .
    JOB-TOTAL-CAL-0.
       ADD 1 TO COUNT.
       ADD JOB-COST (COUNT) TO
       TOTAL-JOB-COST.
       ADD JOB-EXPENSE (COUNT) TO
       TOTAL-EXPENSES.
       ADD JOB-INCOME (COUNT) TO
       TOTAL-INCOME.
       ADD JOB-PROFIT (COUNT) TO
       TOTAL-PROFITS.
    JOB-TOTAL-CAL-1.
       ADD 1 TO COUNT
       ADD JOB-COST (COUNT) TO
       TOTAL-JOB-COST
       ADD JOB-EXPENSE (COUNT), OVERHEAD TO
       TOTAL-EXPENSE
       ADD JOB-INCOME (COUNT) TO
       TOTAL-INCOME
       COMPUTE TOTAL-PROFITS =
       TOTAL-PROFITS + JOB-INCOME (COUNT)
       - JOB-COST (COUNT) - JOB-EXPENSE
       (COUNT) - OVERHEAD.
```

Incorporating Procedures Inline

Based on module size, number of repetitions, modification activities, future expansion considerations, and frequency statistics, small procedures can be moved in-line to minimize overhead requirements.

Tabling

This technique is designed to replace many IF statements by one table look-up statement, or by one computed GO TO statement. For example, if the same data-item is tested in many successive IF statements to set the value of another data-item to some constant, and the range of tested values of the original data-item is limited, then a predetermined table of values could be used to assign the value of the second data-item. Similarly, many consecutive statements of the form

```
IF data-item-1=some-constant GO TO
  some-procedure
```

could be replaced by one computed GO TO statement.

Efficiency Guidelines

Based on execution frequency statistics, the following types of coding inefficiencies may be removed.

1. Unaligned decimal places in arithmetic or numeric comparison operands.
2. Different size operands in moves, comparisons, or arithmetic operations.
3. Mixed usage in arithmetic or numeric comparison operands.
4. Display usage in arithmetic operands or one numeric operand and one display operand in a comparison.
5. SYNC missing for COMP or COMP-1, -2, or -4 items.
6. Inefficient COMP type picture; that is, no sign or more than 9 digits in a COMP item and no sign, even number of digits, or more than 16 digits in COMP-3 items.
7. Noncomputational subscripts.

DIAGNOSTIC MESSAGES

Diagnostic messages are generated by the compiler and listed on SYSLST when errors are found in the source program.

Note: Diagnostic messages (except FIPS diagnostic messages) are suppressed when the NOERRS option is in effect.

WORKING WITH DIAGNOSTIC MESSAGES

1. Approach the diagnostic messages in the order in which they appear on the source listing. It is possible to get compound diagnostic messages. Frequently, an earlier diagnostic message indicates the reason for a later diagnostic message. For example, a missing quotation mark for an alphabetic or alphanumeric literal could involve the inclusion of some clauses not intended for that particular literal. This could cause an apparently valid clause to be diagnosed as invalid because it is not complete, or because it is in conflict with something that preceded it.
2. Check for missing or superfluous punctuation, or other errors of this type.
3. Frequently, a seemingly meaningless message is clarified when the valid syntax or format of the clause or statement in question is referenced.
4. Statement numbers are generated when a verb or procedure-name is encountered.

GENERATION OF DIAGNOSTIC MESSAGES

The compiler scans the statement, element by element, to determine whether the words are combined in a meaningful manner. Based upon the elements that have already been scanned, there are only certain words or elements that can be correctly encountered.

If the anticipated elements are not encountered, a diagnostic message is produced. Some errors may not be uncovered until information from various sections of the program is combined and the inconsistency is noted. Errors uncovered in this manner can produce a slightly different message format than those uncovered when the actual source text is still available. The message that is made unique through that particular error may not contain, for example, the actual source statement that produced the error.

Errors that appear to be identical are diagnosed in a slightly different manner, depending on where they were encountered by the compiler and how they fit within the context of valid syntax. For example, a period missing from the end of the Working-Storage section header is diagnosed specifically as a period required. There is no other information that can appear at that point. However, if at the end of a data item description entry, an element is encountered that is not valid at that point, such as the digits 02, it is diagnosed as invalid. Any clauses associated with the 02 entry which conflict with the clauses in the previous entry (the one that contained the missing period), are diagnosed. Thus, a missing period produces a different type of diagnostic message in one situation than in the other.

If an error occurs during compilation of an ON statement, the diagnostic message may refer to the previous statement number.

Notes:

- If an E-level diagnostic is generated, the LINK option is cancelled, and any linkage editor control statements in the job stream are invalid. For this reason, the following message is issued by the Job Control Processor following the first linkage editor control statement encountered:

```
1S1n D STATEMENT OUT OF SEQUENCE.
     I
```

- If a D-level diagnostic is generated and the error is a compiler error, the job will terminate via the CANCEL macro and produce a dump.

- The following messages will not be issued during a SYNTAX-only compilation or during a CSYNTAX compilation if a C-level error in the diagnostic number ILA0xxx to ILA4xxx range was encountered:

ILA5001I	COMPILER ERROR. COMPILATION ABANDONED.
ILA5002I	COMPILER ERROR. COMPILATION ABANDONED.
ILA5003I	DIVISOR IS ZERO. RESULT WILL BE ALL 9'S.
ILA5004I	ALPHANUMERIC SENDING FIELD TOO BIG. 18 LOW ORDER BYTES USED.

ILA5005I	COMPILER ERROR. COMPILATION ABANDONED.
ILA5006I	COMPILER ERROR. COMPILATION ABANDONED.
ILA5007I	COMPILER ERROR. COMPILATION ABANDONED.
ILA5008I	COMPILER ERROR. COMPILATION ABANDONED.
ILA5009I	COMPILER ERROR. COMPILATION ABANDONED.
ILA5010I	HIGH ORDER TRUNCATION OF THE CONSTANT DID OCCUR.
ILA5011I	HIGH ORDER TRUNCATION MIGHT OCCUR.
ILA5012I	LOST INTERMEDIATE RESULT ATTRIBUTES IN 'XINTR' TABLE. COMPILATION ABANDONED.
ILA5013I	ILLEGAL COMPARISON OF TWO NUMERIC LITERALS. STATEMENT DISCARDED.
ILA5014I	KEY IN SEARCH ALL AT INVALID OFFSET. STATEMENT DISCARDED.
ILA5015I	INVALID USE OF SPECIAL REGISTER. SUBSTITUTING-TALLY.
ILA5016I	MORE THAN 255 SUBSCRIPT ADDRESS CELLS USED. PROGRAM CANNOT EXECUTE CORRECTLY.
ILA5017I	INVALID ADVANCING OPTION FOR A DTFCD FILE. USING STACKER1.
ILA5018I	INTEGER IN POSITIONING OPTION NOT BETWEEN 0 AND 3. 1 ASSUMED.
ILA5019I	PUNCH STACKER SELECT SPECIFIED FOR A DTFPR FILE. USING 'SKIP TO CHANNEL 1'.
ILA5020I	IDENTIFIER NAME(S) IN EXHIBIT EXCEEDS MAXIMUM. TRUNCATED TO 120 CHARACTERS.
ILA5021I	INTEGER IN ADVANCING OR POSITIONING OPTION NOT POSITIVE. POSITIVE ASSUMED.
ILA5022I	MORE THAN 2-DIGIT INTEGER IN ADVANCING OPTION. USING INTEGER 1.
ILA5023I	EOP INVALID FOR DOUBLE-BUFFERED FILE. IGNORED.
ILA5024I	END-OF-PAGE OPTION REQUESTED FOR NON-DTFPR FILE. IGNORED.
ILA5025I	ADVANCING OR POSITIONING OPTION ILLEGAL FOR NON-SEQUENTIAL FILE. IGNORED.
ILA5026I	EXHIBIT OPERAND GREATER THAN 256 BYTES. LENGTH OF 256 ASSUMED.
ILA5027I	NEGATIVE OR ZERO SUBSCRIPT INVALID. CHANGED TO POSITIVE 1.
ILA5028I	RESULT FIELD WILL HAVE POSITIVE SIGN.
ILA5029I	STOP RUN GENERATED AFTER LAST STATEMENT.
ILA5030I	INSTEAD OF AN MVCL INSTRUCTION, AN MVC OR A CALL TO AN OBJECT-TIME SUBROUTINE HAS BEEN GENERATED BECAUSE THE FIELDS OVERLAP DESCRUCTIVELY.
ILA5031I	AN MVCL INSTRUCTION HAS BEEN GENERATED FOR A MOVE INVOLVING AT LEAST ONE LINKAGE SECTION DATA-NAME. IF THE FIELDS OVERLAP DESTRUCTIVELY THE MOVE WILL NOT BE PERFORMED.

In addition, no message of the form ILA6xxx will be issued.

LINKAGE EDITOR OUTPUT

The Linkage Editor produces diagnostic messages, console messages, and a storage map. For a complete description of output and error messages from the Linkage Editor, see the publication DOS/VS System Control Statements. Output resulting from the link editing of a COBOL program is discussed in the chapter "Interpreting Output."

EXECUTION TIME MESSAGES

When an error condition that is recognized by compiler-generated code occurs during execution, an error message is written on SYSLST and often SYSLOG.

Messages that normally appear on SYSLOG are provided with a code indicating from which partition the message originated.

A complete list of execution-time messages can be found in "Appendix I: Diagnostic Messages."

RECORDING PROGRAM STATUS

When a program is expected to run for an extended period of time, provision should be made for taking checkpoint information periodically during the run. A checkpoint is the recording of the status of a problem program and storage (including input/output status and the contents of the general registers). Thus, it provides a means of restarting the job at an intermediate checkpoint position rather than at the beginning, if for any reason processing is terminated before the normal end of the program. For example, a job of higher priority may require immediate processing, or some malfunction (such as a power failure) may occur and cause an interruption. Checkpoints are taken using the COBOL RERUN clause.

Restart is a means of resuming the execution of the program from one of the checkpoints rather than from the beginning of the job. The ability to restart is provided through the RSTRT job control statement. Full details on using this statement are in DOS/VS System Control Statements.

RERUN CLAUSE

The presence of the RERUN clause in the source program causes the CHKPT macro instruction to be issued at the specified interval. When the CHKPT macro instruction is issued, the following information is saved:

1. Information for the Restart and other supervisor or job control routines.
2. The general registers.
3. Bytes 8 through 10, and 12 through 45 of the Communication Region.
4. The problem program area.
5. All file protection extents for files assigned to mass storage devices if the extents are attached to logical units contained in the program for which checkpoints are taken.

Since the COBOL RERUN clause provides a linkage to the system CHKPT macro instruction, any warnings and restrictions on the use of this macro instruction also apply to the use of the RERUN clause. See the publication DOS/VS Supervisor and I/O Macros for a complete description of the CHKPT macro instruction.

TAKING A CHECKPOINT

In order to take a checkpoint, the programmer must specify the source language RERUN clause and must define the file upon which checkpoint records are to be written (for example, ASSGN, EXTEND, etc.). Checkpoin information must be written on a 2311, 2314, 2319, 3330, 3340, 3350, or fixed block mass storage device or on a magnetic tape--either 7- or 9-track. Checkpoint records cannot be embedded in one of the problem program's output files, that is, the program must establish a separate file exclusively for checkpoint records. Checkpoints cannot be written on VSAM files.

In designing a program for which checkpoints are to be taken, the programmer should consider the fact that, upon restarting, the program must be able to continue as though it had just reached that point in the program at which termination occurred. Hence, the programmer should ensure that:

1. File handling is such as to permit easy reconstruction of the status of the system as it existed at the time of checkpoint was taken. For example, when multifile reels are used, the operator should be informed (by message) as to which file is in use at the time a checkpoint is to be taken. He requires this information at restart time.
2. The contents of files are not altered between the time of the checkpoint and the time of the restart. For sequential files, all records written on the file at the time the checkpoint is taken should be unaltered at restart time. For nonsequential files, care must be taken to design the program so that a restart will not duplicate work that has been completed between checkpoint time and restart time. For example, suppose that checkpoint 5 is taken. By adding an amount representing the interest due, account XYZ is updated on a direct-access file that was opened with the I-O option. If the program is restarted from checkpoint 5 and if the interest is recalculated and again added to account XYZ, incorrect results will be produced.

If the program is modular in design, RERUN statements must be included in all modules that handle files for which checkpoints are to be taken. (When an entry point of a module containing a RERUN statement is encountered, a COBOL subroutine, ILBDCKP0, is called. ILBDCKP0 enters the files of the module into the list of files to be repositioned.) Repositioning to the proper record will not occur for any files that were defined in modules other than those containing RERUN statements. Moreover, a restart from any given checkpoint may not reposition other tapes on which checkpoints are stored. Note, too, that only one disk checkpoint file can be used.

RESTARTING A PROGRAM

If the programmer requests checkpoints in his job by means of the COBOL RERUN clause, the following message is given each time a checkpoint is taken:

```
0C001 CHKPT nnnn HAS BEEN TAKEN ON
      SYSxxx
```

nnnn
: is the 4-character identification of the checkpoint record.

To restart a job from a checkpoint, the following steps are required:

1. Replace the // EXEC statement with a // RSTRT statement. The format of the RSTRT statement is discussed in the chapter "Preparing COBOL Programs For Processing." All other job control statements applicable to the job step should be the same as when the job was originally run. If necessary, the channel and unit addresses for the // ASSGN control statements may be changed.
2. Rewind all tapes used by the program being restarted, and mount them on devices assigned to the symbolic units required by the program. If multivolume files are used, mount (on the primary unit) the reel being used at the time that the checkpoint was taken, and rewind it. If multifile volumes are used, position the reel to the start of the file referenced at the time the checkpoint is being taken.
3. Reposition any card file so that only cards not yet read when the checkpoint was taken are in the card reader.
4. Execute the job.
5. A checkpointed program can be restarted only in the same partition. The virtual partition must start at the same location as when the program was checkpointed and its end address must not be lower than at that time. This is because checkpoint dumps the entire virtual partition.

The following is a sample COBOL program and the output listing resulting from its compilation, link editing, and execution. The program creates a blocked, unlabeled, standard sequential file, writes it out on tape, and then reads it back in. It also does a check on the field called NO-OF-DEPENDENTS. All data records in the file are displayed. Those with a zero in the NO-OF-DEPENDENTS field are displayed with the special character Z. The records of the file are not altered from the time of creation, despite the fact that the NO-OF-DEPENDENTS field is changed for display purposes. The individual records of the file are created using the subscripting technique.

The output formats illustrated in the listing are described in the chapter "Interpreting Output."

```
// JOB SAMPLE
// OPTION NODECK,LINK,LIST,LISTX,SYM,ERRS
// EXEC FCOBOL
```

```
    1  IBM DOS VS COBOL                         REL 1.0          PP NO. 5746-CB1                  08.17.32  10/02/73

 CBL QUOTE,OPT,SXREF,LVL=A
00001    000010 IDENTIFICATION DIVISION.
00002    000020 PROGRAM-ID. TESTRUN.
00003                  AUTHOR.  PROGRAMMER NAME.
00004                  INSTALLATION.  NEW YORK DEVELOPMENT CENTER
00005                  DATE-WRITTEN.  APRIL 18,1973
00006           DATE-COMPILED. 10/02/73
00007           REMARKS.  THIS PROGRAM HAS BEEN WRITTEN AS A SAMPLE PROGRAM FOR
00008               COBOL USERS.  IT CREATES AN OUTPUT FILE AND READS IT BACK
00009               AS INPUT.
00010    000100
00011    000110 ENVIRONMENT DIVISION.
00012    000120 CONFIGURATION SECTION.
00013    000130 SOURCE-COMPUTER. IBM-360-H50.
00014    000140 OBJECT-COMPUTER. IBM-370.
00015    000150 INPUT-OUTPUT SECTION.
00016    000160 FILE-CONTROL.
00017    000170     SELECT FILE-1 ASSIGN TO SYS008-UT-2400-S.
00018    000180     SELECT FILE-2 ASSIGN TO SYS008-UT-2400-S.
00019    000190
00020    000200 DATA DIVISION.
00021    000210 FILE SECTION.
00022    000220 FD  FILE-1
00023    000230     LABEL RECORDS ARE OMITTED
00024    000240     BLOCK CONTAINS 5 RECORDS
00025    000250     RECORDING MODE IS F
00026    000255     RECORD CONTAINS 20 CHARACTERS
00027    000260     DATA RECORD IS RECORD-1.
00028    000270 01  RECORD-1.
00029               05  FIELD-A PIC X(20).
00030    000290 FD  FILE-2
00031    000300     LABEL RECORDS ARE OMITTED
00032    000310     BLOCK CONTAINS 5 RECORDS
00033    000320     RECORD CONTAINS 20 CHARACTERS
00034    000330     RECORDING MODE IS F
00035    000340     DATA RECORD IS RECORD-2.
00036    000350 01  RECORD-2.
00037               05  FIELD-A PIC X(20).
```

```
000370 WORKING-STORAGE SECTION.
000380 01  FILLER.
           02  KOUNT PIC S99 COMP SYNC.
000400     02  ALPHABET PIC X(26) VALUE IS "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
000410     02  ALPHA REDEFINES ALPHABET PIC X OCCURS 26 TIMES.
000420     02  NUMBR PIC S99 COMP SYNC.
000430     02  DEPENDENTS PIC X(26) VALUE "01234012340123401234012340".
000440     02  DEPEND REDEFINES DEPENDENTS PIC X OCCURS 26 TIMES.
000450 01  WORK-RECORD.
000460     05  NAME-FIELD PIC X.
000470     05  FILLER PIC X VALUE IS SPACE.
000480     05  RECORD-NO PIC 9999.
000490     05  FILLER PIC X VALUE IS SPACE.
000500     05  LOCATION PIC AAA VALUE IS "NYC".
000510     05  FILLER PIC X VALUE IS SPACE.
000520     05  NO-OF-DEPENDENTS PIC XX.
000530     05  FILLER PIC X(7) VALUE IS SPACES.
000540
000550 PROCEDURE DIVISION.
       BEGIN.
000570     NOTE THAT THE FOLLOWING OPENS THE OUTPUT FILE TO BE CREATED
000580     AND INITIALIZES COUNTERS.
000590 STEP-1.  OPEN OUTPUT FILE-1.  MOVE ZERO TO KOUNT, NUMBR.
000600     NOTE THAT THE FOLLOWING CREATES INTERNALLY THE RECORDS TO BE
000610     CONTAINED IN THE FILE, WRITES THEM ON TAPE, AND DISPLAYS
000620     THEM ON THE CONSOLE.
000630 STEP-2.  ADD 1 TO KOUNT, NUMBR. MOVE ALPHA (KOUNT) TO
000640     NAME-FIELD.
000650     MOVE DEPEND (KOUNT) TO NO-OF-DEPENDENTS
000660     MOVE NUMBR TO RECORD-NO.
000670 STEP-3. DISPLAY WORK-RECORD UPON CONSOLE. WRITE RECORD-1 FROM
000680     WORK-RECORD.
000690 STEP-4.  PERFORM STEP-2 THRU STEP-3 UNTIL KOUNT IS EQUAL TO 26.
000700     NOTE THAT THE FOLLOWING CLOSES THE OUTPUT FILE AND REOPENS
000710     IT AS INPUT.
000720 STEP-5. CLOSE FILE-1. OPEN INPUT FILE-2.
000730     NOTE THAT THE FOLLOWING READS BACK THE FILE AND SINGLES
000740     OUT EMPLOYEES WITH NO DEPENDENTS.
000750 STEP-6. READ FILE-2 RECORD INTO WORK-RECORD AT END GO TO STEP-8.
000760 STEP-7. IF NO-OF-DEPENDENTS IS EQUAL TO "0" MOVE "Z" TO
000770     NO-OF-DEPENDENTS. EXHIBIT NAMED WORK-RECORD. GO TO STEP-6.
000780 STEP-8. CLOSE FILE-2.
000790     STOP RUN.
```

```
IBM DOS VS COBOL                         REL 1.0              PP NO. 5746-CB1           08.17.32  10/02/73

INTRNL NAME    LVL SOURCE NAME                       BASE    DISPL   INTRNL NAME    DEFINITION    USAGE        R  O  Q  M


DNM=1-148      FD  FILE-1                           DTF=01           DNM=1-148                    DTFMT                  F
DNM=1-179      01  RECORD-1                          BL=1     000    DNM=1-179      DS 0CL20      GROUP
DNM=1-200      02  FIELD-A                           BL=1     000    DNM=1-200      DS 20C        DISP
DNM=1-217      FD  FILE-2                           DTF=02           DNM=1-217                    DTFMT                  F
DNM=1-248      01  RECORD-2                          BL=2     000    DNM=1-248      DS 0CL20      GROUP
DNM=1-269      02  FIELD-A                           BL=2     000    DNM=1-269      DS 20C        DISP
DNM=1-289      01  FILLER                            BL=3     000    DNM=1-289      DS 0CL56      GROUP
DNM=1-308      02  KOUNT                             BL=3     000    DNM=1-308      DS 1H         COMP
DNM=1-323      02  ALPHABET                          BL=3     002    DNM=1-323      DS 26C        DISP
DNM=1-341      02  ALPHA                             BL=3     002    DNM=1-341      DS 1C         DISP         R  O
DNM=1-359      02  NUMBR                             BL=3     01C    DNM=1-359      DS 1H         COMP
DNM=1-374      02  DEPENDENTS                        BL=3     01E    DNM=1-374      DS 26C        DISP
DNM=1-394      02  DEPEND                            BL=3     01E    DNM=1-394      DS 1C         DISP         R  O
DNM=1-410      01  WORK-RECORD                       BL=3     038    DNM=1-410      DS 0CL20      GROUP
DNM=1-434      02  NAME-FIELD                        BL=3     038    DNM=1-434      DS 1C         DISP
DNM=1-454      02  FILLER                            BL=3     039    DNM=1-454      DS 1C         DISP
DNM=1-473      02  RECORD-NO                         BL=3     03A    DNM=1-473      DS 4C         DISP-NM
DNM=1-492      02  FILLER                            BL=3     03E    DNM=1-492      DS 1C         DISP
DNM=2-000      02  LOCATION                          BL=3     03F    DNM=2-000      DS 3C         DISP
DNM=2-018      02  FILLER                            BL=3     042    DNM=2-018      DS 1C         DISP
DNM=2-037      02  NO-OF-DEPENDENTS                  BL=3     043    DNM=2-037      DS 2C         DISP
DNM=2-063      02  FILLER                            BL=3     045    DNM=2-063      DS 7C         DISP
```

```
                MEMORY MAP

                     TGT                      003F8

                SAVE AREA                     003F8
                SWITCH                        00440
                TALLY                         00444
                SORT SAVE                     00448
                ENTRY-SAVE                    0044C
                SORT CORE SIZE                00450
                NSTD-REELS                    00454
                SORT RET                      00456
                WORKING CELLS                 00458
                SORT FILE SIZE                00588
                SORT MODE SIZE                0058C
                PGT-VN TBL                    00590
                TGT-VN TBL                    00594
                SORTAB ADDRESS                00598
                LENGTH OF VN TBL              0059C
                LNGTH OF SORTAB               0059E
                PGM ID                        005A0
                A(INIT1)                      005A8
                UPSI SWITCHES                 005AC
                DEBUG TABLE PTR               005B4
                CURRENT PRIORITY              005B8
                TA LENGTH                     005B9
                PRBL1 CELL PTR                005BC
                UNUSED                        005C0
                RESERVED                      005C4
                VSAM SAVE AREA ADDRESS        005C8
                UNUSED                        005CC
                RESERVED                      005D4
                OVERFLOW CELLS                005EC
                BL CELLS                      005EC
                DTFADR CELLS                  005F8
                FIB CELLS                     00600
                TEMP STORAGE                  00608
                TEMP STORAGE-2                00610
                TEMP STORAGE-3                00610
                TEMP STORAGE-4                00610
                BLL CELLS                     00610
                VLC CELLS                     00614
                SBL CELLS                     00614
                INDEX CELLS                   00614
                SUBADR CELLS                  00614
                ONCTL CELLS                   0061C
                PFMCTL CELLS                  0061C
                PFMSAV CELLS                  0061C
                VN CELLS                      00620
                SAVE AREA =2                  00624
                XSASW CELLS                   00624
                XSA CELLS                     00624
                PARAM CELLS                   00624
                RPTSAV AREA                   00628
                CHECKPT CTR                   00628
                IOPTR CELLS                   00628
                DEBUG TABLE                   00628
```

```
LITERAL POOL (HEX)

00640 (LIT+0)      00000001  001A5B5B  C2D6D7C5  D5405B5B  C2C3D3D6  E2C55B5B
00658 (LIT+24)     C2C6C3D4  E4D35B5B  C0000000

     DISPLAY LITERALS (BCD)

00664 (LTL+36)    'WORK-RECORD'


                     PGT                       00628

                DEBUG LINKAGE AREA             00628
                OVERFLOW CELLS                 00628
                VIRTUAL CELLS                  0062C
                PROCEDURE NAME CELLS           00638
                GENERATED NAME CELLS           00638
                SUBDTF ADDRESS CELLS           0063C
                VNI CELLS                      0063C
                LITERALS                       00640
                DISPLAY LITERALS               00664
                PROCEDURE BLOCK CELLS          00670


 REGISTER ASSIGNMENT

  REG 6      BL =3
  REG 7      BL =1
  REG 8      BL =2
WORKING-STORAGE STARTS AT LOCATION 00100 FOR A LENGTH OF 00050.

 PROCEDURE BLOCK ASSIGNMENT

  PBL = REG 11

  PBL =1     STARTS AT LOCATION 000674  STATEMENT 60
```

```
0
57
            000674                          PN=02    EQU    *
60
            000674                          PN=03    EQU    *
60
            000674                          START    EQU    *
            000674  58 B0 C 048                      L      11,048(0,12)         PBL=1
            000678  58 20 D 1F4                      L      2,1F4(0,13)          BL =1
            00067C  41 10 C 01E                      LA     1,01E(0,12)          LIT+6
            000680  58 00 D 200                      L      0,200(0,13)          DTF=1
            000684  18 40                            LR     4,0
            000686  05 F0                            BALR   15,0
            000688  50 00 F 008                      ST     0,008(0,15)
            00068C  45 00 F 00C                      BAL    0,00C(0,15)
            000690  00000000                         DC     X'00000000'
            000694  0A 02                            SVC    2
            000696  41 00 D 200                      LA     0,200(0,13)          DTF=1
            00069A  58 F0 C 008                      L      15,008(0,12)         V(ILBDIMLO)
            00069E  05 EF                            BALR   14,15
            0006A0  58 10 D 200                      L      1,200(0,13)          DTF=1
            0006A4  96 10 1 020                      OI     020(1),X'10'
            0006A8  50 20 D 1F4                      ST     2,1F4(0,13)          BL =1
            0006AC  58 70 D 1F4                      L      7,1F4(0,13)          BL =1
60
            0006B0  D2 01 6 000 C 018                MVC    000(2,6),018(12)     DNM=1-308       LIT+0
            0006B6  D2 01 6 01C C 018                MVC    01C(2,6),018(12)     DNM=1-359       LIT+0
64
            0006BC                          PN=04    EQU    *
64
            0006BC  48 30 C 01A                      LH     3,01A(0,12)          LIT+2
            0006C0  4A 30 6 000                      AH     3,000(0,6)           DNM=1-308
            0006C4  4E 30 D 210                      CVD    3,210(0,13)          TS=01
            0006C8  D7 05 D 210 D 210                XC     210(6,13),210(13)    TS=01           TS=01
            0006CE  94 0F D 216                      NI     216(13),X'0F'        TS=01+6
            0006D2  4F 30 D 210                      CVB    3,210(0,13)          TS=01
            0006D6  40 30 6 000                      STH    3,000(0,6)           DNM=1-308
            0006DA  48 30 C 01A                      LH     3,01A(0,12)          LIT+2
            0006DE  4A 30 6 01C                      AH     3,01C(0,6)           DNM=1-359
            0006E2  4E 30 D 210                      CVD    3,210(0,13)          TS=01
            0006E6  D7 05 D 210 D 210                XC     210(6,13),210(13)    TS=01           TS=01
            0006EC  94 0F D 216                      NI     216(13),X'0F'        TS=01+6
            0006F0  4F 30 D 210                      CVB    3,210(0,13)          TS=01
            0006F4  40 30 6 01C                      STH    3,01C(0,6)           DNM=1-359
64
            0006F8  41 40 6 002                      LA     4,002(0,6)           DNM=1-341
            0006FC  48 20 6 000                      LH     2,000(0,6)           DNM=1-308
            000700  4C 20 C 01A                      MH     2,01A(0,12)          LIT+2
            000704  1A 42                            AR     4,2
            000706  5B 40 C 018                      S      4,018(0,12)          LIT+0
            00070A  50 40 D 21C                      ST     4,21C(0,13)          SBS=1
            00070E  58 E0 D 21C                      L      14,21C(0,13)         SBS=1
            000712  D2 00 6 038 E 000                MVC    038(1,6),000(14)     DNM=1-434       DNM=1-341
            000718  41 40 6 01E                      LA     4,01E(0,6)           DNM=1-394
            00071C  48 20 6 000                      LH     2,000(0,6)           DNM=1-308
            000720  4C 20 C 01A                      MH     2,01A(0,12)          LIT+2
            000724  1A 42                            AR     4,2
            000726  5B 40 C 018                      S      4,018(0,12)          LIT+0
            00072A  50 40 D 220                      ST     4,220(0,13)          SBS=2
            00072E  58 F0 D 220                      L      15,220(0,13)         SBS=2
            000732  D2 00 6 043 F 000                MVC    043(1,6),000(15)     DNM=2-37        DNM=1-394
            000738  92 40 6 044                      MVI    044(6),X'40'         DNM=2-37+1
67
            00073C  48 30 6 01C                      LH     3,01C(0,6)           DNM=1-359
            000740  4E 30 D 210                      CVD    3,210(0,13)          TS=01
            000744  F3 31 6 03A D 216                UNPK   03A(4,6),216(2,13)   DNM=1-473       TS=07
            00074A  96 F0 6 03D                      OI     03D(6),X'F0'         DNM=1-473+3
```

```
68
               00074E                        PN=05     EQU    *
68
               00074E   58 F0 C 00C                    L      15,00C(0,12)          V(ILBDDSP0)
               000752   05 1F                          BALR   1,15
               000754   0002                           DC     X'0002'
               000756   00                             DC     X'00'
               000757   000014                         DC     X'000014'
               00075A   0D0001FC                       DC     X'0D0001FC'           BL =3
               00075E   0038                           DC     X'0038'
               000760   FFFF                           DC     X'FFFF'
68
               000762   D2 13 7 000 6 038              MVC    000(20,7),038(6)      DNM=1-179        DNM=1-410
               000768   58 10 D 200                    L      1,200(0,13)           DTF=1
               00076C   18 41                          LR     4,1
               00076E   58 F0 1 010                    L      15,010(0,1)
               000772   45 E0 F 00C                    BAL    14,00C(0,15)
               000776   50 20 D 1F4                    ST     2,1F4(0,13)           BL =1
               00077A   58 70 D 1F4                    L      7,1F4(0,13)           BL =1
               00077E   58 10 D 228                    L      1,228(0,13)           VN=01
               000782   07 F1                          BCR    15,1
70
               000784                        PN=06     EQU    *
70
               000784   D2 03 D 224 D 228              MVC    224(4,13),228(13)     PSV=1            VN=01
               00078A   41 00 B 11E                    LA     0,11E(0,11)           GN=01
               00078E   50 00 D 228                    ST     0,228(0,13)           VN=01
               000792                        GN=01     EQU    *
               000792   48 30 6 000                    LH     3,000(0,6)            DNM=1-308
               000796   49 30 C 01C                    CH     3,01C(0,12)           LIT+4
               00079A   47 80 B 12E                    BC     8,12E(0,11)           GN=02
               00079E   47 F0 B 048                    BC     15,048(0,11)          PN=04
               0007A2                        GN=02     EQU    *
               0007A2   D2 03 D 228 D 224              MVC    228(4,13),224(13)     VN=01            PSV=1
73
               0007A8                        PN=07     EQU    *
73
               0007A8   58 10 D 200                    L      1,200(0,13)           DTF=1
               0007AC   94 EF 1 020                    NI     020(1),X'EF'
               0007B0   18 01                          LR     0,1
               0007B2   18 40                          LR     4,0
               0007B4   41 10 C 026                    LA     1,026(0,12)           LIT+14
               0007B8   07 00                          BCR    0,0
               0007BA   05 F0                          BALR   15,0
               0007BC   50 00 F 008                    ST     0,008(0,15)
               0007C0   45 00 F 00C                    BAL    0,00C(0,15)
               0007C4   00000000                       DC     X'00000000'
               0007C8   0A 02                          SVC    2
               0007CA   58 00 D 200                    L      0,200(0,13)           DTF=1
               0007CE   41 10 C 02E                    LA     1,02E(0,12)           LIT+22
               0007D2   0A 02                          SVC    2
73
               0007D4   41 10 C 01E                    LA     1,01E(0,12)           LIT+6
               0007D8   58 00 D 204                    L      0,204(0,13)           DTF=2
               0007DC   18 40                          LR     4,0
               0007DE   05 F0                          BALR   15,0
               0007E0   50 00 F 008                    ST     0,008(0,15)
               0007E4   45 00 F 00C                    BAL    0,00C(0,15)
               0007E8   00000000                       DC     X'00000000'
               0007EC   0A 02                          SVC    2
               0007EE   41 00 D 204                    LA     0,204(0,13)           DTF=2
               0007F2   58 F0 C 008                    L      15,008(0,12)          V(ILBDIML0)
               0007F6   05 EF                          BALR   14,15
               0007F8   58 10 D 204                    L      1,204(0,13)           DTF=2
               0007FC   96 10 1 020                    OI     020(1),X'10'
```

```
 76
            000800                      PN=08    EQU    *
 76
            000800  58 10 D 204                  L      1,204(0,13)       DTF=2
            000804  91 20 1 010                  TM     010(1),X'20'
            000808  47 10 B 1BE                  BC     1,1BE(0,11)       GN=03
            00080C  18 41                        LR     4,1
            00080E  41 F0 C 010                  LA     15,010(0,12)      GN=03
            000812  D2 02 1 025 F 001            MVC    025(3,1),001(15)
            000818  58 F0 1 010                  L      15,010(0,1)
            00081C  45 E0 F 008                  BAL    14,008(0,15)
            000820  50 20 D 1F8                  ST     2,1F8(0,13)       BL =2
            000824  58 80 D 1F8                  L      8,1F8(0,13)       BL =2
            000828  D2 13 6 038 8 000            MVC    038(20,6),000(8)  DNM=1-410        DNM=1-248
            00082E  47 F0 B 1C2                  BC     15,1C2(0,11)      GN=04
            000832                      GN=03    EQU    *
 76
            000832  47 F0 B 208                  BC     15,208(0,11)      PN=010
            000836                      GN=04    EQU    *
 77
            000836                      PN=09    EQU    *
 77
            000836  95 F0 6 043                  CLI    043(6),X'F0'      DNM=2-37
            00083A  47 70 B 1DA                  BC     7,1DA(0,11)       GN=05
            00083E  95 40 6 044                  CLI    044(6),X'40'      DNM=2-37+1
 77         000842  47 70 B 1DA                  BC     7,1DA(0,11)       GN=05
            000846  92 E9 6 043                  MVI    043(6),X'E9'      DNM=2-37
            00084A  92 40 6 044                  MVI    044(6),X'40'      DNM=2-37+1
            00084E                      GN=05    EQU    *
 78
            00084E  58 10 C 038                  L      1,038(0,12)       LIT+32
            000852  50 10 D 22C                  ST     1,22C(0,13)       PRM=1
            000856  41 20 D 22C                  LA     2,22C(0,13)       PRM=1
            00085A  58 F0 C 00C                  L      15,00C(0,12)      V(ILBDDSP0)
            00085E  05 1F                        BALR   1,15
            000860  8001                         DC     X'8001'
            000862  10                           DC     X'10'
            000863  00000B                       DC     X'00000B'
            000866  0C00003C                     DC     X'0C00003C'       LIT+36
            00086A  0000                         DC     X'0000'
            00086C  00                           DC     X'00'
            00086D  000014                       DC     X'000014'
            000870  0D0001FC                     DC     X'0D0001FC'       BL =3
            000874  0038                         DC     X'0038'
            000876  FFFF                         DC     X'FFFF'
 78
            000878  47 F0 B 18C                  BC     15,18C(0,11)      PN=08
 79
            00087C                      PN=010   EQU    *
 79
            00087C  58 10 D 204                  L      1,204(0,13)       DTF=2
            000880  94 EF 1 020                  NI     020(1),X'EF'
            000884  18 01                        LR     0,1
            000886  18 40                        LR     4,0
            000888  41 10 C 026                  LA     1,026(0,12)       LIT+14
            00088C  07 00                        BCR    0,0
            00088E  05 F0                        BALR   15,0
            000890  50 00 F 008                  ST     0,008(0,15)
            000894  45 00 F 00C                  BAL    0,00C(0,15)
            000898  00000000                     DC     X'00000000'
            00089C  0A 02                        SVC    2
            00089E  58 00 D 204                  L      0,204(0,13)       DTF=2
            0008A2  41 10 C 02E                  LA     1,02E(0,12)       LIT+22
            0008A6  0A 02                        SVC    2
 80
            0008A8  0A 0E                        SVC    14
```

```
0008AA  50 D0 5 008           INIT2    ST     13,008(0,5)
0008AE  50 50 D 004                    ST     5,004(0,13)
0008B2  58 20 C 004                    L      2,004(0,12)          VIR=1
0008B6  95 00 2 000                    CLI    000(2),X'00'
0008BA  07 79                          BCR    7,9
0008BC  92 FF 2 000                    MVI    000(2),X'FF'
0008C0  96 10 D 048                    OI     048(13),X'10'        SWT+0
0008C4  50 E0 D 054           INIT3    ST     14,054(0,13)
0008C8  05 F0                          BALR   15,0
0008CA  91 20 D 048                    TM     048(13),X'20'        SWT+0
0008CE  47 E0 F 016                    BC     14,016(0,15)
0008D2  58 00 B 048                    L      0,048(0,11)
0008D6  98 2D B 050                    LM     2,13,050(11)
0008DA  58 E0 D 054                    L      14,054(0,13)
0008DE  07 FE                          BCR    15,14
0008E0  96 20 D 048                    OI     048(13),X'20'        SWT+0
0008E4  41 60 0 004                    LA     6,004(0,0)
0008E8  41 10 C 000                    LA     1,000(0,12)
0008EC  41 70 C 003                    LA     7,003(0,12)          VIR=1-1
0008F0  05 50                          BALR   5,0
0008F2  58 40 1 000                    L      4,000(0,1)
0008F6  1E 4B                          ALR    4,11
0008F8  50 40 1 000                    ST     4,000(0,1)
0008FC  87 16 5 000                    BXLE   1,6,000(5)
000900  41 10 C 010                    LA     1,010(0,12)          PN=01
000904  41 70 C 017                    LA     7,017(0,12)          LIT+0-1
000908  05 50                          BALR   5,0
00090A  58 40 1 000                    L      4,000(0,1)
00090E  1E 4B                          ALR    4,11
000910  50 40 1 000                    ST     4,000(0,1)
000914  87 16 5 000                    BXLE   1,6,000(5)
000918  41 80 D 1F4                    LA     8,1F4(0,13)          OVF=1
00091C  41 70 D 20F                    LA     7,20F(0,13)          TS=01-1
000920  05 10                          BALR   1,0
000922  58 00 8 000                    L      0,000(0,8)
000926  1E 0B                          ALR    0,11
000928  50 00 8 000                    ST     0,000(0,8)
00092C  87 86 1 000                    BXLE   8,6,000(1)
000930  58 60 D 1FC                    L      6,1FC(0,13)          BL =3
000934  58 70 D 1F4                    L      7,1F4(0,13)          BL =1
000938  58 80 D 1F8                    L      8,1F8(0,13)          BL =2
00093C  D2 03 D 228 C 014              MVC    228(4,13),014(12)    VN=01               VNI=1
000942  58 E0 D 1B0                    L      14,1B0(0,13)
000946  90 6D E 060                    STM    6,13,060(14)
00094A  58 E0 D 054                    L      14,054(0,13)
00094E  07 FE                          BCR    15,14
000000  05 F0                 INIT1    BALR   15,0
000002  07 00                          BCR    0,0
000004  90 0E F 00A                    STM    0,14,00A(15)
000008  47 F0 F 082                    BC     15,082(0,15)
00000C  00000000                       DC     30F'0'
000084  58 C0 F 0C6                    L      12,0C6(0,15)
000088  58 E0 C 004                    L      14,004(0,12)         VIR=1
00008C  58 D0 F 0CA                    L      13,0CA(0,15)
000090  95 00 E 000                    CLI    000(14),X'00'
000094  47 70 F 0A2                    BC     7,0A2(0,15)
000098  96 10 D 048                    OI     048(13),X'10'        SWT+0
00009C  92 FF E 000                    MVI    000(14),X'FF'
0000A0  47 F0 F 0AC                    BC     15,0AC(0,15)
```

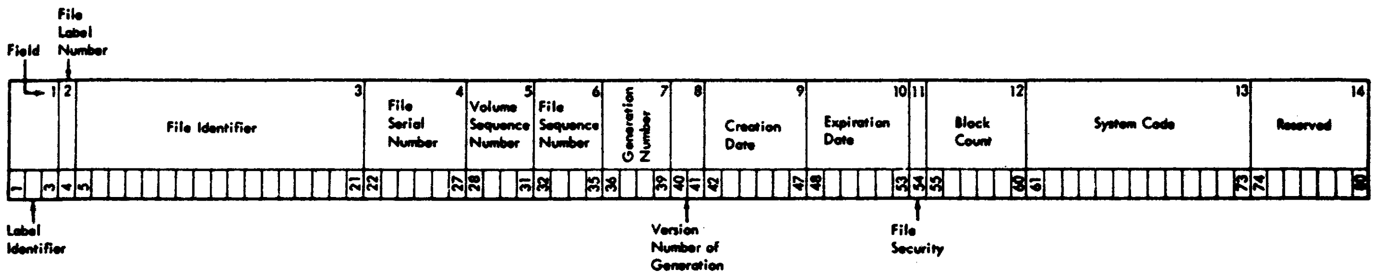
```
              0000A4   98 CE F 03A                 LM     12,14,03A(15)
              0000A8   90 EC D 00C                 STM    14,12,00C(13)
              0000AC   18 5D                       LR     5,13
              0000AE   98 9F F 0BA                 LM     9,15,0BA(15)
              0000B2   91 10 D 048                 TM     048(13),X'10'          SWT+0
              0000B6   07 19                       BCR    1,9
              0000B8   07 FF                       BCR    15,15
              0000BA   07 00                       BCR    0,0
              0000BC   000008C4                    ADCON  L4(INIT3)
              0000C0   00000000                    ADCON  L4(INIT1)
              0000C4   00000000                    ADCON  L4(INIT1)
              0000C8   00000628                    ADCON  L4(PGT)
              0000CC   000003F8                    ADCON  L4(TGT)
              0000D0   00000674                    ADCON  L4(START)
              0000D4   000008AA                    ADCON  L4(INIT2)
              0000D8   C3D6C2D6F3F0F0F0            DC     X'C3D6C2D6F3F0F0F0'
              0000E0   E3C5E2E3D9E4D540            DC     X'E3C5E2E3D9E4D540'
              0000E8   00000000                    DC     X'00000000'
              0000EC   F1F061F0F261F7F3            DC     X'F1F061F0F261F7F3'
              0000F4   F0F84BF1F74BF3F2            DC     X'F0F84BF1F74BF3F2'




*STATISTICS*          SOURCE RECORDS =     80      DATA ITEMS =      22      NO OF VERBS =     28
*STATISTICS*          PARTITION SIZE = 655176      LINE COUNT =      56      BUFFER SIZE =    512
*OPTIONS IN EFFECT*   PMAP RELOC ADR =   NONE      SPACING    =      1       FLOW       =    NONE
*OPTIONS IN EFFECT*      LISTX         QUOTE       SYM    NOCATALR        LIST         LINK       NOSTXIT        NOLIB
*OPTIONS IN EFFECT*     NOCLIST        FLAGW       ZWB    NOSUPMAP        XREF         ERRS         SXREF          OPT
*OPTIONS IN EFFECT*     NOSTATE        TRUNC       SEQ    NOSYMDMP       NODECK       NOVERB      NOSYNTAX        LVL=A
```

```
     IBM DOS VS COBOL                          REL 1.0              PP NO. 5746-CB1          08.17.32  10/02/73

                                                    CROSS-REFERENCE DICTIONARY

   DATA NAMES                      DEFN     REFERENCE


   ALPHA                           000042   000064
   ALPHABET                        000041
   DEPEND                          000045   000066
   DEPENDENTS                      000044
   FIELD-A                         000029
   FIELD-A                         000037
   FILE-1                          000017   000060   000068   000073
   FILE-2                          000018   000073   000076   000079
   KCUNT                           000040   000060   000064   000066   000070
   LOCATION                        000051
   NAME-FIELD                      000047   000064
   NO-OF-DEPENDENTS                000053   000066   000077
   NUMBR                           000043   000060   000064   000067
   RECORD-NO                       000049   000067
   RECORD-1                        000028   000068
   RECORD-2                        000036   000076
   WORK-RECORD                     000046   000068   000076   000078




   PROCEDURE NAMES                 DEFN     REFERENCE


   BEGIN                           000057
   STEP-1                          000060
   STEP-2                          000064   000070
   STEP-3                          000068   000070
   STEP-4                          000070
   STEP-5                          000073
   STEP-6                          000076   000078
   STEP-7                          000077
   STEP-8                          000079   000076



   CARD   ERROR MESSAGE

   00064  ILA5011I-W    HIGH ORDER TRUNCATION MIGHT OCCUR.
   00064  ILA5011I-W    HIGH ORDER TRUNCATION MIGHT OCCUR.

                         FEDERAL INFORMATION PROCESSING STANDARDS (FIPS) DIAGNOSTIC MESSAGES                  PAGE    1

  LINE   NUMBER          MESSAGE

  00006  ILA8003I-W      DATE-COMPILED PARAGRAPH IS AN EXTENSION TO FIPS LEVEL A.
  00025  ILA8002I-W      RECORDING MODE IS CLAUSE IS AN EXTENSION TO ALL FIPS LEVELS.
  00034  ILA8002I-W      RECORDING MODE IS CLAUSE IS AN EXTENSION TO ALL FIPS LEVELS.
  00054  ILA8003I-W      SPACES IS AN EXTENSION TO FIPS LEVEL A.
  00060  ILA8003I-W      COMMA OR SEMICOLON AS PUNCTUATION IS AN EXTENSION TO FIPS LEVEL A.
  00062  ILA8003I-W      COMMA OR SEMICOLON AS PUNCTUATION IS AN EXTENSION TO FIPS LEVEL A.
  00062  ILA8003I-W      COMMA OR SEMICOLON AS PUNCTUATION IS AN EXTENSION TO FIPS LEVEL A.
  00064  ILA8003I-W      COMMA OR SEMICOLON AS PUNCTUATION IS AN EXTENSION TO FIPS LEVEL A.
  00064  ILA8003I-W      MULTIPLE RESULTS IN ADD STATEMENT IS AN EXTENSION TO FIPS LEVEL A.
  00068  ILA8003I-W      UPON OPTION OF DISPLAY STATEMENT IS AN EXTENSION TO FIPS LEVEL A.
  00068  ILA8002I-W      UPON CONSOLE  OPTION OF DISPLAY STATEMENT IS AN EXTENSION TO ALL LEVELS.
  00068  ILA8003I-W      FROM OPTION OF WRITE STATEMENT IS AN EXTENSION TO FIPS LEVEL A.
  00070  ILA8003I-W      UNTIL OPTION OF PERFORM STATEMENT IS AN EXTENSION TO FIPS LEVEL A.
  00076  ILA8003I-W      INTO OPTION OF READ STATEMENT IS AN EXTENSION TO FIPS LEVEL A.
  00078  ILA8002I-W      EXHIBIT STATEMENT IS AN EXTENSION TO ALL FIPS LEVELS.

  END OF COMPILATION
```

```
// EXEC LNKEDT




JOB  SAMPLE                DOS LINKAGE EDITOR DIAGNOSTIC OF INPUT

ACTION TAKEN  MAP REL
LIST   AUTOLINK   IJFFBZZN
LIST   AUTOLINK   ILBDDSP0
LIST   AUTOLINK   IJJCPDV
LIST   AUTOLINK   ILBDDSS0
LIST    INCLUDE IJJCPDV
LIST   AUTOLINK   ILBDIML0
LIST   AUTOLINK   ILBDMNS0
LIST   AUTOLINK   ILBDSAE0
LIST   ENTRY




10/02/73   PHASE  XFR-AD  LOCORE  HICORE  DSK-AD    ESD TYPE  LABEL     LOADED  REL-FR

        PHASE***  07D878  07D878  07F1FF  05F 0F 4  CSECT     TESTRUN   07D878  07D878  RELOCATABLE

                                                    CSECT     IJFFBZZN  07E1C8  07E1C8
                                                    *  ENTRY  IJFFZZZN  07E1C8
                                                    *  ENTRY  IJFFBZZZ  07E1C8
                                                    *  ENTRY  IJFFZZZZ  07E1C8

                                                    CSECT     ILBDSAE0  07F078  07F078
                                                       ENTRY  ILBDSAE1  07F0C0

                                                    CSECT     ILBDMNS0  07F070  07F070

                                                    CSECT     ILBDIML0  07F018  07F018

                                                    CSECT     ILBDDSP0  07E578  07E578
                                                       ENTRY  ILBDDSP1  07E978

                                                    CSECT     ILBDDSS0  07ECF0  07ECF0
                                                       ENTRY  ILBDDSS1  07EF50
                                                       ENTRY  ILBDDSS2  07EF48
                                                       ENTRY  ILBDDSS3  07F008
                                                       ENTRY  ILBDDSS4  07ED16
                                                       ENTRY  ILBDDSS5  07EDC2
                                                       ENTRY  ILBDDSS6  07EE22
                                                       ENTRY  ILBDDSS7  07EDEC
                                                       ENTRY  ILBDDSS8  07ED46

                                                    CSECT     IJJCPDV   07EAA8  07EAA8
                                                       ENTRY  IJJCPDV1  07EAA8
                                                    *  ENTRY  IJJCPDV2  07EAA8

* UNREFERENCED SYMBOLS                              WXTRN     STXITPSW
                                                    WXTRN     ILBDDBG2

002 UNRESOLVED ADDRESS CONSTANTS
```

```
// ASSGN   SYS008,X'483'
// EXEC

WORK-RECORD = A 0001 NYC Z
WORK-RECORD = B 0002 NYC 1
WORK-RECORD = C 0003 NYC 2
WORK-RECORD = D 0004 NYC 3
WORK-RECORD = E 0005 NYC 4
WORK-RECORD = F 0006 NYC Z
WORK-RECORD = G 0007 NYC 1
WORK-RECORD = H 0008 NYC 2
WORK-RECORD = I 0009 NYC 3
WORK-RECORD = J 0010 NYC 4
WORK-RECORD = K 0011 NYC Z
WORK-RECORD = L 0012 NYC 1
WORK-RECORD = M 0013 NYC 2
WORK-RECORD = N 0014 NYC 3
WORK-RECORD = O 0015 NYC 4
WORK-RECORD = P 0016 NYC Z
WORK-RECORD = Q 0017 NYC 1
WORK-RECORD = R 0018 NYC 2
WORK-RECORD = S 0019 NYC 3
WORK-RECORD = T 0020 NYC 4
WORK-RECORD = U 0021 NYC Z
WORK-RECORD = V 0022 NYC 1
WORK-RECORD = W 0023 NYC 2
WORK-RECORD = X 0024 NYC 3
WORK-RECORD = Y 0025 NYC 4
WORK-RECORD = Z 0026 NYC Z


EOJ SAMPLE
```

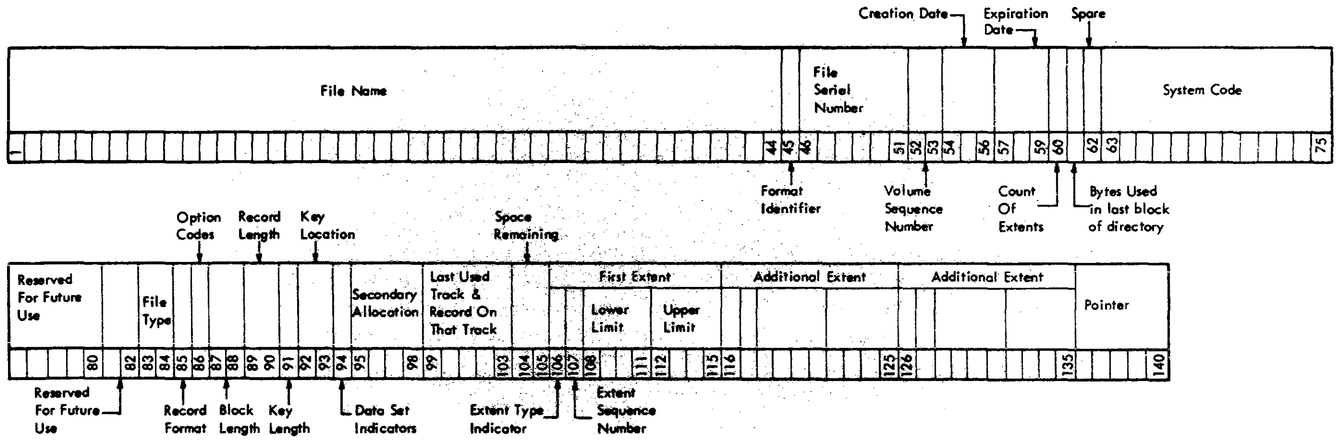
```
BG
BG A 0001 NYC 0
BG B 0002 NYC 1
BG C 0003 NYC 2
BG D 0004 NYC 3
BG E 0005 NYC 4
BG F 0006 NYC 0
BG G 0007 NYC 1
BG H 0008 NYC 2
BG I 0009 NYC 3
BG J 0010 NYC 4
BG K 0011 NYC 0
BG L 0012 NYC 1
BG M 0013 NYC 2          ③
BG N 0014 NYC 3
BG O 0015 NYC 4
BG P 0016 NYC 0
BG Q 0017 NYC 1
BG R 0018 NYC 2
BG S 0019 NYC 3
BG T 0020 NYC 4
BG U 0021 NYC 0
BG V 0022 NYC 1
BG W 0023 NYC 2
BG X 0024 NYC 3
BG Y 0025 NYC 4
BG Z 0026 NYC 0
BG EOJ SAMPLE
   00.56.19,DURATION 00.03.42
```



The standard tape file label format and contents are as follows:

Field	Name and Length	Description
1.	<u>LABEL IDENTIFIER</u> 3 bytes, EBCDIC	Identifies the type of label. HDR = Header (beginning of a data file) EOF = End-of-file (end of a set of data) EOV = End-of-volume (end of the physical reel)
2.	<u>FILE LABEL NUMBER</u> 1 byte, EBCDIC	Always a 1.
3.	<u>FILE IDENTIFIER</u> 17 bytes, EBCDIC	Uniquely identifies the entire file, may contain only printable characters. Some other systems will not accept embedded blanks in the file identifier.
4.	<u>FILE SERIAL NUMBER</u> 6 bytes, EBCDIC	Uniquely identifies a file/volume relationship. This field is identical to the volume serial number in the volume label of the first or only volume of a multivolume file or a multifile set. This field will normally be numeric (000001 to 999999), but may contain any six alphanumeric characters.
5.	<u>VOLUME SEQUENCE NUMBER</u> 4 bytes	Indicates the order of a volume in a given file or multifile set. The first must be numbered 0001, and subsequent numbers must be in proper numeric sequence.
6.	<u>FILE SEQUENCE</u> 4 bytes	Assigns numeric sequence to a file within a multifile set. The first must be numbered 0001.
7.	<u>GENERATION TIME</u> 4 bytes	Uniquely identifies the various editions of the file. May be from 0001 to 9999 in proper numeric sequence.
8.	<u>VERSION NUMBER OF GENERATION</u> 2 bytes	Indicates the version of a generation of a file.

Field	Name and Length	Description
9.	CREATION DATE 6 bytes	Indicates the year and the day of the year that the file was created. Position / Code / Meaning 1 / blank / none 2-3 / 00-99 / year 4-6 / 001-366 / day of year (e.g., January 31, 1973 would be entered as 73031).
10.	EXPIRATION DATE 6 bytes	Indicates the year and the day of the year when the file may become a scratch tape. The format of this field is identical to field 9. On a multifile reel processed sequentially, all files are considered to expire on the same day.
11.	FILE SECURITY 1 byte	Indicates security status of the file. 0 = No security protection. 1 = Security protection. Additional identification of the file is required before it can be processed.
12.	BLOCK COUNT 6 bytes	Indicates the number of data blocks written in the file from the last header label to the first trailer label, exclusive of tapemarks. Count does not include checkpoint records. This field is used in trailer labels.
13.	SYSTEM CODE 13 bytes	Uniquely identifies the operating system.
14.	RESERVED 7 bytes	Reserved. Should be recorded as blanks.



Format 1: This format is common to all data files on disk.

Field	Name and Length	Description
1.	FILE NAME 44 bytes, alphanumeric EBCDIC	This field serves as the key portion of the file label. It can consist of three sections:

1. File ID is an alphanumeric field assigned by the programmer and identifies the file. It can be 1 through 35 bytes in length if generation and version numbers are used, or 1 through 44 bytes in length if they are not used.

2. Generation Number. If used, this field is separated from File ID by a period. It has the format Gnnnn, where G identifies the field as the generation number and nnnn (in decimal) identifies the generation of the file.

3. Version Number of Generation. If used, this section immediately follows the generation number and has the format Vnn, where V identifies the field as the version of generation number and nn (in decimal) identifies the version of generation of the file.

Note: IBM DOS/VS System compares the entire field against the filename given in the DLBL card. The generation and version numbers are treated differently by the IBM OS/VS System.

Fields 2 through 33 constitute the DATA portion of the file label.

Field	Name and Length	Description
2.	FORMAT IDENTIFIER 1 byte, EBCDIC numeric	1 = format 1
3.	FILE SERIAL NUMBER 6 bytes, alphanumeric EBCDIC	Uniquely identifies a file/volume relationship. It is identical to the volume serial number of the first or only volume of a multivolume file.
4.	VOLUME SEQUENCE NUMBER 2 bytes, binary	Indicates the order of a volume relative to the first volume on which the data file resides.
5.	CREATION DATE 3 bytes, discontinuous binary	Indicates the year and the day of the year the file was created. It is of the form YDD, where Y signifies the year (0-99) and DD the day of the year (1-366).
6.	EXPIRATION DATE 3 bytes, discontinuous binary	Indicates the year and the day of the year the file may be deleted. The form of this field is identical to that of field 5.
7a.	EXTENT COUNT 1 byte, binary	Contains a count of the number of extents for this file on this volume. If user labels are used, the count includes the user label track as a separate extent. This field is maintained by the Disk Operating System.
7b.	BYTES USED IN LAST BLOCK OF DIRECTORY 1 byte, binary	Used by IBM Operating System Virtual Storage only for partitioned (library structure) data sets. Not used by the Disk Operating System Virtual Storage.
7c.	SPARE 1 byte	Reserved for future use.
8.	SYSTEM CODE 13 bytes	Uniquely identifies the operating system.
9.	RESERVED 7 bytes	Reserved for future use.
10.	FILE TYPE 2 bytes	The contents of this field uniquely identify the type of data file.

Hex Code	Meaning
4000	Sequential organization
2000	Direct organization
8000	Indexed organization
0200	Library organization
0000	Organization not defined in the file label

Field	Name and Length	Description
11.	RECORD FORMAT 1 byte	The contents of this field indicate the type of records contained in the file.

Bit Position	Content	Meaning
0 and 1	01	Variable-length records
	10	Fixed-length records
	11	Undefined format
2	0	No track overflow
	1	File is organized using track overflow (IBM OS/VS only)
3	0	Unblocked records
	1	Blocked records
4	0	No truncated records
	1	Truncated records in file
5 and 6	01	Control character ASA code
	10	Control character machine code
	00	Control character not stated
7	0	Records are written without keys
	1	Records are written with keys

Field	Name and Length	Description
12.	OPTION CODES 1 byte	Bits within this field are used to indicate various options used in building the file.

Bit Position	Meaning
0	If on, indicates data file was created using write validity check.
1-7	Unused.

Field	Name and Length	Description
13.	BLOCK LENGTH 2 bytes, binary	Indicates the block length for fixed-length records, or maximum block size for variable-length blocks.
14.	RECORD LENGTH 2 bytes, binary	Indicates the record length for fixed-length records, or the maximum record length for variable-length records.
15.	KEY LENGTH 1 byte, binary	Indicates the length of the key portion of the data records in the file.
16.	KEY LOCATION 2 bytes, binary	Indicates the high-order position of the data record.

Field	Name and Length	Description
17.	DATA SET INDICATORS 1 byte	Bits within this field are used to indicate the following:

Bit Position	Meaning
0	If on, indicates that this is the last volume on which this file normally resides. This bit is used by the DOS/VS DTFSR routine only. None of the other bits in this byte are used by the DOS/VS.
1	If on, indicates that the data set described by this file must remain in the same absolute location on the direct-access device.
2	If on, indicates that block length must always be a multiple of eight bytes.
3	If on, indicates that this data file is security protected; a password must be provided in order to access it.
4-7	Space. Reserved for future use.

Field	Name and Length	Description
18.	SECONDARY ALLOCATION 4 bytes, binary	Indicates the amount of storage to be requested for this data file at end-of-extent. This field is used by IBM OS/VS only. It is not used by DOS/VS routines.
19.	LAST USED TRACK AND RECORD ON THAT TRACK 5 bytes, discontinuous binary	Indicates the last occupied track in a consecutive file organization data file. This field has the format CCHHR. It is all binary zeros if the last track in a consecutive data file is not on this volume, or if it is not consecutive organization.
20.	AMOUNT OF SPACE REMAINING ON LAST TRACK USED 2 bytes, binary	A count of the number of bytes of available space remaining on the last track used by this data file on this volume.
21.	EXTENT TYPE INDICATOR 1 byte	Indicates the type of extent with which the following fields are associated:

Hex Code	Meaning
00	Next three fields do not indicate any extent.
01	Prime area (indexed) or consecutive area, etc., (i.e., the extent containing the user's data records).
02	Overflow area of an indexed file.
04	Cylinder index or master index area of an indexed file.
40	User label track area.
80	Shared cylinder indicator.

Field	Name and Length	Description
22.	EXTENT SEQUENCE NUMBER 1 byte, binary	Indicates the extent sequence in a multi-extent file.
23.	LOWER LIMIT 4 bytes, discontinuous binary	The cylinder and the track address specifying the starting point (lower limit) of this extent component. This field has the format CCHH.
24.	UPPER LIMIT 4 bytes	The cylinder and the track address specifying the end point (upper limit) of this extent component. This field has the format CCHH.
25-28.	ADDITIONAL EXTENT 10 bytes	These fields have the same format as the fields 21 through 24, above.
29-32.	ADDITIONAL EXTENT 10 bytes	These fields have the same format as fields 21 through 24, above.
33.	POINTER TO NEXT FILE LABEL WITHIN THIS LABEL SET 5 bytes, discontinuous binary	The disk address (format CCHHR) of a continuation label is needed to further describe the file. If field 9 indicates indexed organization, this field will point to a Format 2 file label within this label set. Otherwise, it points to a Format 3 file label, and then only if the file contains more than three extent segments. If no additional file label is pointed to, this field contains all binary zeros.

The track format for the 2311, 2314, 2319, 2321, 3330, 3340, and 3350 direct-access storage devices is illustrated in Figure 67. The names of the fields are given in the following discussion.

Index Marker: All tracks start with an index marker. It is a signal to the hardware that indicates beginning of the track.

Home Address: The home address, preceded by a gap, follows the index marker. The home address uniquely identifies each track by specifying the cylinder and head number.

Track Descriptor Record (Record 0): Record 0 consists of two parts: a count portion and a data portion. The count portion is the same as it is for any other record (see the following description of count for record 1. The 8-byte data portion is used to record information used by LIOCS. The information in the data portion depends on the data organization (direct or indexed) that is being used.

For direct organization, this portion in the form of CCHHR contains the address of the last record on the track and the number of bytes remaining on the track. This information is used to determine whether there is space for another record on the track. For indexed organization, the data portion contains the address of the last record in the cylinder overflow area and the number of tracks remaining in the cylinder overflow area. Record 0 is then used as the cylinder overflow control record.

Address Marker: All records after record 0 will be preceded by a 2-byte address marker. The address marker is a signal to the hardware that a record is starting.

Data Records: Data records can consist of a count and data portion for sequential organization, or a count, key, and data portion for direct and indexed organizations.

1. Count Portion. The count portion contains the identification of each record, the key length, and the data length.

   a. Identification. Each record is identified with its cylinder number, head number, or record number. The cylinder and head numbers will be the same as those of the home address. The record number will indicate a particular record on the track. That is, the first record after record 0 will be record 1, followed by record 2, etc. This 5-byte binary field in the form of CCHHR is often referred to as the record ID.

   b. Key Length. The key length is specified in an 8-bit byte; its length can range from 0 to 255. This field will contain a zero if there is no key.

   c. Data Length. The data length is specified in the 16 bits of the next two bytes.

   Note: It is the count portion that identifies the presence or absence of a key, in addition to indicating the data length. In this way, each record is unique and self formatting.

2. Key Portion. The key portion of the record is normally used to store the control field of the data record such as a man number. Direct and indexed files must have a key portion.

3. Data Portion. The data portion of the record contains the data record.

Note that all records, including the data record, terminate with a 2-byte cyclic check. The hardware uses this cyclic check to ensure that is correctly reread what it had written. The cyclic check is cumulative and is appended to each record when it is written. Upon reading the record, the cyclic check is again accumulated and then compared with the appended cyclic check. If they do not agree, a data check is initiated.

The first byte of the count portion of each record and the home address is reserved for a flag byte. If a track becomes defective, a utility program may be used to transfer the data to an alternate track. (Cylinders 200 through 202 are reserved for alternate tracks on the 2321. Strips 6 through 9 of subcell 19 of each cell are reserved for alternate tracks on the 2321.) In this case, a flag bit within the byte is set on to indicate that this is a defective track and the address of an alternate track will be placed in the record ID of record 0. Subsequent references to this defective track will result in the Supervisor accessing record 0 for the address of the alternate track.

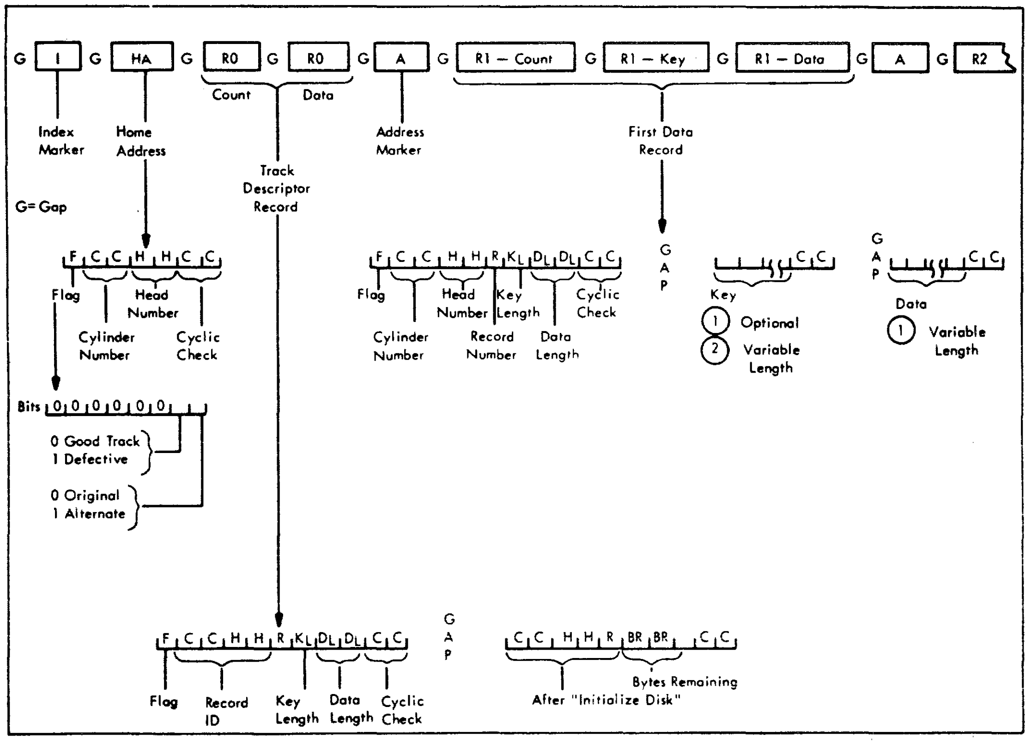


Figure 67. Track Format

The IBM DOS/VS COBOL Object-Time Subroutine Library, Program Number 5746-LM4, is packaged with the DOS/VS COBOL Compiler and also available as a separate product. It provides subroutines to be link edited with object modules produced by DOS/VS COBOL Compiler. It also provides subroutines that can be dynamically fetched during problem program execution.

There are several major categories of COBOL library subroutines:

- Input/output verb routines
- ASCII support routines
- Conversion routines
- Arithmetic verb routines
- Sort/Merge Feature interface routines
- Checkpoint (RERUN) routines
- Segmentation Feature routines
- Other verb routines
- Object-time debugging routines
- Object-time execution statistics routines
- Optimizer routines
- Transient routines

The following sections describe some of the more commonly used subroutines.

INPUT/OUTPUT SUBROUTINES

The input/output subroutines are used for the COBOL verbs DISPLAY (TRACE and EXHIBIT), ACCEPT, STOP (literal), READ, WRITE, REWRITE, OPEN, CLOSE, DELETE, and START printer spacing, printer overflow, input/output errors, disk formatting and extent handling, and tape and sequential disk labels.

Printer Spacing

The ILBDSPA0 subroutine is used to control printer spacing when the WRITE statement with the BEFORE/AFTER ADVANCING or POSITIONING option is specified in the source program.

Tape and Sequential Disk Labels

The ILBDUSL0 and ILBDNSL0 subroutines are used when user or nonstandard labels, respectively, are to be processed (LABEL RECORDS ARE data-name).

CLOSE WITH LOCK Subroutine

The ILBDCLK0 subroutine is given control on an OPEN if the file is ever closed with lock in the program. It checks whether the OPEN statement is used to open a file previously closed with lock. If the file was previously closed with lock, it issues an object-time message and terminates the current job.

WRITE Statement Subroutines

The ILBDVBL0 subroutine is used to write variable-length blocked records.

The ILBDDIO0 subroutine is used for writing files with direct organization (DTFDA).

The ILBDISM0 subroutine is used for writing files with indexed organization.

READ Statement Subroutines

The ILBDDSR0 subroutine is used to read sequentially the records of a directly organized file.

The ILBDDIO0 subroutine is used to read randomly the records of a directly organized file.

The ILBDISM0 subroutine is used to read an indexed file.

REWRITE Statement Subroutines

The ILBDDIO0 subroutine is used to update records on a directly organized file.

The ILBDISM0 subroutine is used to update an indexed file.

DISPLAY (EXHIBIT and TRACE) Subroutines

The ILBDDSP0 subroutine formats one or more operands into printed lines, performing conversions as needed.

The ILBDOSY0 and ILBDASY0 subroutines open SYSLST and/or SYSPCH and/or SYSIPT if there are DISPLAY or ACCEPT statements in a label declarative.

ACCEPT and STOP (literal) Statement Subroutines

The ILBDACP0 subroutine is used to handle ACCEPT statements for both SYSIPT and the console, as well as the STOP (literal) statement. The ILBDACP0 subroutine does not format or convert operands. For operands greater than 80 characters in length, any remainder in excess of the nearest multiple of 80 is ignored when accepting data from SYSIPT.

CLOSE Subroutine

The ILBDCRD0 subroutine is given control when a CLOSE UNIT statement is issued for a sequential input file with direct organization.

Multiple File Tape Subroutine

The ILBDMFT0 subroutine is given control when a reel contains more than one file and there are no standard labels.

Tape Pointer Subroutine

The ILBDIML0 subroutine locates the pointer to the physical tape drive associated with the logical unit for a particular tape file.

Input/Output Error Subroutines

The ILBDSAE0 subroutine is used for processing input/output errors that occur on tape and sequential disk.

The ILBDDAE0 subroutine is used for processing input/output errors that occur on directly organized files.

The ILBDISE0 subroutine is called whenever an input/output error occurs during the processing an indexed file.

The ILBDABX0 subroutine is used to issue a STXIT macro instruction causing control to be passed to it if there is an error on a unit-record device.

Disk Extent Subroutines

The ILBDFMT0 subroutine writes record 0 (R0) on each track of each extent of a directly organized file opened as output, and writes an end-of-file (EOF) record as the last record in the file. This subroutine is called after the file has been opened.

The ILBDXTN0 subroutine stores for subsequent use the extent information for directly organized files.

3886 OCR Subroutine

The ILBDOCR0 subroutine is used to perform I/O operations for the 3886 Optical Character Reader.

VSAM Subroutines

The ILBDINT0 subroutine does initialization for VSAM processing.

The ILBDVOC0 performs VSAM open and close functions.

The ILBDVIO0 performs all action requests for VSAM files (for example, READ, WRITE, REWRITE, START, DELETE).

These routines may call the Checkpoint subroutine and $$BCOBR1 discussed later in this chapter.

Auxiliary Subroutines

Certain input/output subroutines use auxiliary subroutines as follows:

Auxiliary Routine	Used By
ILBDMOV0	ILBDSPA0, ILBDNSL0, ILBDVBL0
ILBDIDA0	ILBDFMT0, ILBDDSR0
ILBDTAB0	ILBDDIO0, ILBDIDA0, ILBDCKP0

ASCII SUPPORT SUBROUTINES

The subroutine described below handles functions necessary for files written in ASCII. Other functions are handled by code generated by the compiler or by the subroutine ILBDSPA0.

Separately Signed Numeric Subroutine

The ILBDSSN0 subroutine is called to check the validity of signs described as TRAILING SEPARATE CHARACTER or LEADING SEPARATE CHARACTER.

CONVERSION SUBROUTINES

Eight numeric data formats are permitted in COBOL: five external (for input and output) and three internal (for internal processing).

The five external formats are:

- External or zoned decimal
- External floating-point
- Sterling display
- Numeric edited
- Sterling report

The three internal formats are:

- Internal or packed decimal
- Binary
- Internal floating-point

The conversions from internal decimal to external decimal, from external decimal to internal decimal, and from internal decimal to numeric edited are performed in-line. The other conversions are performed by the COBOL library subroutines shown in Table 35.

Table 35. Functions of COBOL Library Conversion Subroutines

Subroutine Name and Entry Points	Conversion	
	From	To
ILBDEFL2	External floating-point	Internal decimal
ILBDEFL1	External floating-point	Binary
ILBDEFL0	External floating-point	Internal floating-point
ILBDBID0[1] ILBDBID1[1] ILBDBID2[1]	Binary	Internal decimal
ILBDBIE0[1] ILBDBIE1[1] ILBDBIE2[1]	Binary	External decimal
ILBDBII0[2] ILBDBII1[2]	Binary	Internal floating-point
ILBDTEF0[2] ILBDTEF1[2]	Binary	External floating-point
ILBDTEF2	Internal decimal	External floating-point
IFBDTEF3	Internal floating-point	External floating-point
ILBDIDB0	Internal decimal	Binary
ILBDIDB1	External decimal	Binary
ILBDDCI1	Internal decimal	Internal floating-point
ILBDDCI0	External decimal	Internal floating-point
ILBDIFD0	Internal floating-point	Internal decimal
ILBDIFD1	Internal floating-point	External decimal
ILBDIFB1 ILBDIFB2[3]	Internal floating-point	Binary integer and a power of 10 exponent
ILBDIFB0[3]	Internal floating-point	Binary
ILBDIDR0	Internal decimal	Sterling report
ILBDIDT0	Internal decimal	Sterling non-report
ILBDSTI0	Sterling non-report	Internal decimal

[1]The entry points used depend on whether the double-precision number is in registers 0 and 1, 2 and 3, or 4 and 5, respectively.
[2]The entry points are for single-precision binary and double-precision binary, respectively.
[3]This entry point is used for calls from other COBOL library subroutines.

ARITHMETIC VERB SUBROUTINES

Most arithmetic operations are performed in-line. However, involved calculations with very large numbers, such as decimal multiplication of two 30-digit numbers, are performed by COBOL library arithmetic subroutines. These subroutine names and their functions are shown in Table 36.

SORT/MERGE FEATURE INTERFACE ROUTINE

Communication between the Sort/Merge program and the COBOL program is maintained by ILBDSRT0 and ILBDMRG0.

CHECKPOINT (RERUN) SUBROUTINE

The ILBDCKP0 subroutine issues the checkpoint macro instruction, which will write checkpoint records on a programmer-specified tape or disk checkpoint device. There are two calling sequences to this subroutine. The first, ILBDCKP1, is activated during initialization when the addresses of all files in the program are entered in a table. The second, ILBDCKP2, is required to take checkpoints during a sorting operation.

If RERUN is requested during a sorting operation, ILBDSRT0 must gather a list of physical IOCS files in use by the Sort program every time Sort exits at E11, E21, and E31. ILBDSRT0 then calls the checkpoint subroutine which will take a checkpoint of all active files.

SEGMENTATION FEATURE SUBROUTINE

The Segmentation Feature requires an object time subroutine, ILBDSEM0. The ILBDSEM0 subroutine performs the following functions when segments are needed:

1. Loads and initializes independent segments not in storage.
2. Loads overlayable segments not in storage.
3. Initializes independent segments if the segment is in storage.
4. Branches to desired entry points.

OTHER VERB ROUTINES

There are also COBOL library subroutines for comparisons, the verbs MOVE and TRANSFORM, and other features of the COBOL language.

Compare Subroutines

The ILBDVCO0 subroutine compares two operands, one or both of which is variable in length. Each may exceed 256 bytes.

The ILBDIVL0 subroutine is used in comparisons involving the figurative constant ALL 'literal', where literal is greater than one character.

Table 36. Functions of COBOL Library Arithmetic Subroutines

Subroutine Name	Function
ILBDXMU0	Internal decimal multiplication (30 digits * 30 digits = 60 digits)
ILBDXDI0	Internal decimal division (60 digits/30 digits = 30 digits)
ILBDXPR0	Decimal fixed-point exponentiation
ILBDFPW0	Floating-point exponentiation
ILBDGPW0[1]	Floating-point exponentiation

[1]The ILBDGPW0 entry point is used if the exponent has a PICTURE clause specifying an integer. The ILBDFPW0 entry point is used in all other cases.

MOVE Subroutines

The ILBDVMO0 subroutine is used when one or both operands is variable in length and in-line instructions cannot be generated (for example, fields overlap, etc). Each may exceed 256 bytes. The subroutine has two entry points, depending on the type of MOVE: ILBDVMO0 (left-justified) and ILBDVMO1 (right-justified).

The ILBDANF0 subroutine is used to move the figurative constant ALL 'literal', where literal is greater than one character.

The ILBDANE0 subroutine is used to perform a right-or left-justified alphanumeric edited move.

The ILBDSMV0 subroutine handles moves to right-justified receiving fields either greater than 512 bytes in length or variable in length.

TRANSFORM Subroutine

The ILBDVTR0 subroutine transforms variable-length items using the ILBDTRN0 transform table.

Class Test Subroutine

The ILBDCLS0 subroutine is used to perform class tests for variable-length items and those fixed-length items longer than 256 bytes.

Note: The following tables are placed in the library for use by the in-line coding generated by the compiler and the subroutines called for by both the class test and TRANSFORM:

ILBDATB0	-- Alphabetic class test
ILBDETB0	-- External decimal class test
ILBDITB0	-- Internal decimal class test
ILBDUTB0	-- Unsigned internal decimal
ILBDWTB0	-- Unsigned external decimal

SEARCH Subroutine

The ILBDSCH0 subroutine processes each search argument key according to type.

Main Program or Subprogram Subroutine

The ILBDMNS0 subroutine is a 1-byte switch tested in the code generated for EXIT PROGRAM, GOBACK, INIT1, and INIT2.

The ILBDSET0 subroutine must be called by a non-American National Standard COBOL program prior to any call to an American National Standard COBOL program. When calling ILBDSET0, standard linkage conventions must be observed; there are no parameters to be passed. The ILBDSET0 subroutine sets the 1-byte switch (ILBDMNS0) to X'FF'. This switch is tested in the American National Standard COBOL program to determine whether it is a main or a called program. The name of this subroutine can be changed to any name desired by the COBOL user.

OBJECT-TIME DEBUGGING SUBROUTINES

Three options are available for object-time debugging. These are the statement number option (STATE), the flow trace option (FLOW), and the symbolic debug option (SYMDMP). The subroutines for the first two options provide debugging information when a program terminates abnormally; the subroutines for the third option provide debugging information either at abnormal termination or dynamically during execution of a program. All of the subroutines are under the control of and are serviced by the Debug Control Subroutine (ILBDDBG0). This section discusses (1) the Debug Control Subroutine, and (2) the subroutines that are called in response to each of the three debugging options.

Debug Control Subroutine

The ILBDDBG0 subroutine is included in the load module whenever the CBL control card for a program contains at least one of the debugging options, or when the CBL control card for a program requests execution statistics.

Statement Number Subroutine

The ILBDSTN0 subroutine provides the number of the statement and the number of the verb being executed when abnormal termination occurs. If abnormal termination occurs during execution of an instruction outside of the COBOL program, the statement number that is provided is that of the last COBOL instruction executed.

Flow Trace Subroutine

Space is allocated at compile time for a flow trace table using the programmer-specified number in the FLOW option of the CBL card. (If FLOW=0 was specified for a subprogram, no space is allocated; rather the subprogram shares the table space reserved by that program preceding it in the calling sequence for which a FLOW specification was made.)

Each time the flow trace subroutine ILBDFLW0 receives control from the COBOL program, it inserts the executing program's identification as well as the card number of the current procedure into the next available position in the table. When the end of the table is reached, subsequent entries overlay the first set of entries. The procedure is repeated until the end of the program or until abnormal termination. If abnormal termination occurs, the subroutine produces a list of each entry of the table, beginning with the earliest entry.

Symbolic Debug Subroutines

The symbolic debug subroutines provide a formatted symbolic dump, either dynamically at execution time, or at abnormal termination.

The following subroutines perform initialization and prccess debug control cards:

ILBDMP1C, ILBDMP11, ILBDMP12, ILBDMP13, ILBDMP14, and either ILBDMP01 or ILBDMP02.

To provide a dump at abnormal termination, the following subroutines are used:

ILBDMP20, ILBDMP21, ILBDMP22, ILBDMP23, ILBDMP24, ILBDMP25 and ILBDMP01 or ILBDMP02. These subroutines are not included in the load module at link edit time; they are loaded dynamically during program execution.

The ILBDADR0 subroutine tests the validity of an address calculated for a subscripted identifier or the validity of the starting and ending addresses of a variable-length identifier used as the receiving field in a MOVE statement.

OBJECT-TIME EXECUTION STATISTICS SUBROUTINES

The object-time execution statistics subroutines enable the printing of execution statistics when a program terminates normally (via STOP RUN or GOBACK in the main program) and when a program terminates abnormally. In addition, when COUNT is requested, the debug control subroutine (described above) is also included in the load module.

COUNT Initialization Subroutine

The ILBDTC00 subroutine is called from the debug control subroutine to get space for an initialize the table and chains which service the COUNT options.

COUNT Frequency Subroutine

The ILBDCT10 subroutine maintains the execution frequency statistics.

COUNT Termination Subroutine

The ILBDTC20 subroutine is included in all COBOL load modules. It determines if execution frequency statistics were requested.

COUNT Print Subroutine

The ILBDTC30 subroutine formats and prints the execution frequency statistics.

OPTIMIZER SUBROUTINES

GO TO ... DEPENDING ON Subroutine

The ILBDGD00 subroutine is called only when the optimization option (OPT) has been specified. It is used to more efficiently process GO TO statements with the DEPENDING ON option in both segmented and nonsegmented programs.

Optimizer DISPLAY Subroutine

The ILBDDSS0 subroutine is used to print or type certain data types on SYSLST or the console, respectively.

TRANSIENT SUBROUTINES

The IBM DOS/VS COBOL Object-Time Subroutine Library includes routines that are dynamically fetched during program execution. These routines are as follows:

Symbolic Debug Subroutines

With the exception of ILBDDBG0, the symbolic debug subroutines described previously are transient routines.

SYMDMP Error Message Subroutine

The $$BCOBEM subroutine prepares SYMDMP error messages.

Error Message Subroutine

The $$BCOBER subroutine prepares input/output error messages.

Error Message Print Subroutine

The $$BCOBR1 subroutine prints the error messages prepared by $$BCOBER and provides a dump if the DUMP option is in effect.

Reposition Tape Subroutine

The $$BFCMUL subroutine resets the PUB pointer for a particular (SYSnnn) device to the same as that saved earlier by the subroutine ILBDIML0.

Note: If dynamically fetched subroutines are required during problem program execution, the Subroutine Library must be installed on the object machine. If dynamically fetched subroutines are not required during problem program execution, the object-time subroutines can be link edited on the source machine; the Subroutine Library must in this case be installed on the source machine.

This appendix contains information concerning system and size requirements for the DOS/VS COBOL compiler, execution time considerations, and the Sort/Merge Feature. Additional information used in estimating the virtual and auxiliary storage requirements is contained in the publication IBM DOS/VS COBOL Compiler and Library, Installation Reference Material.

MINIMUM MACHINE REQUIREMENTS FOR THE COMPILER

1. A System/370 supported by DOS/VS. A minimum of 60K bytes of virtual storage is required.

2. Five work files. The system logical unit SYSLNK must be assigned to a single area (extent) on a 2314, 2319, 3330, 3340, 3350, or fixed block mass storage device. Four programmer logical units (SYS001 through SYS004) must reside on 2400, 3410, 3420 tape units, or on 2314, 2319, 3330, 3340, 3350, or fixed block mass storage devices. At least one programmer logical unit as well as the operating system must reside on a mass storage device (that is, a 2311, 2314, 2319, 3330, 3340, 3350, or fixed block mass storage device). If the three remaining logical units reside on tape, there must be a separate tape unit for each file. If they reside on mass storage devices, there must be enough space on those devices. An additional logical unit, SYS005, must be assigned if the symbolic debug option (SYMDMP) is being used. Logical unit SYS006 must be assigned for the FIPS flagger.

   Work file assignments must be made as follows:

   ```
   SYSLNK - mass storage device
   SYS001 - mass storage device
   SYS002 - mass storage device or tape
            unit
   SYS003 - mass storage device or tape
            unit
   SYS004 - mass storage device or tape
            unit
   SYS005 - mass storage device or tape
            unit
   SYS006 - mass storage device unit
   ```

   Note that SYSLNK need only be assigned at compile time if the CATAL or LINK option is in effect.

   The filenames for SYSLNK and SYS001 through SYS006 on the TLBL or DLBL statements are IJSYSLN, IJSYS01, IJSYS02, IJSYS03, IJSYS04, IJSYS05, and IJSYS06, respectively. If the "filename" parameter of the SYMDMP option is specified, this filename is used instead of IJSYS05 on DLBL statements.

3. A device, such as a printer keyboard, for direct operator communication.

4. A device, such as a card reader, for the job input stream.

5. A device, such as a printer or tape unit, for system output files.

6. The floating-point arithmetic feature, if floating-point literals or calculations are used.

SOURCE PROGRAM SIZE CONSIDERATIONS

Compiler Capacity

This section contains information which must be considered in determining the limitations on the size of a COBOL source program in a specific virtual storage size. It also contains information to aid the programmer in determining how his source program affects usage of space at compilation time.

The capacity of the COBOL compiler is limited by two general conditions: (1) the total table requirement may be greater than the space available and (2) the fact that an individual table (with the exception of the ADCON and cross-reference tables) may need to be longer than 32,767 bytes. If either of these conditions are met during compilation, one of the following error messages will be issued:

ILA0001I-D	NO MORE TABLE SPACE AVAILABLE. COMPILATION ABANDONED.
ILA0003I-D	A TABLE HAS EXCEEDED THE MAXIMUM SIZE. COMPILATION ABANDONED.
ILA6007I-D	TABLE HAS EXCEEDED MAXIMUM SIZE. LISTX, OBJECT MODULE, AND DECK WILL BE INCOMPLETE. INCREASE PARTITION.

In each case, compilation is terminated. However, in the first and third cases, or in the case of overflow of the ADCON or cross-reference table, the program may be recompiled with a larger size parameter.

The compiler will accept and compile a 1500 card program in the minimum storage of 64K. In this configuration, the minimum size compiler input/output areas must be allocated. If both LINK and DECK are specified, more storage is required for buffer space, which reduces the space available to a given program. Within this configuration, the compiler will accept programs much larger than 1500 cards; the specific size limitation for any storage size depends entirely on the statement mix in that program, but the limiting factors are described in the next section.

The overall critical limit using the minimum buffer specification may be expressed as follows:

2 (number of pn's + gn's + literals + virtuals) + 8A + S (L + 5D + 8V + 3P) $\leq$ 14336 + C

where the number of virtuals is the number of calls to COBOL object-time subroutine entry points and subprograms specified in a CALL statement, and V is the number of unique such names; also

A = number of entries in the ADCON table as defined below

S = 1 if the Segmentation Feature is required and NOOPT is in effect; otherwise 0

L = length of optimized literals

D = number of segment discontiguities in the Procedure Division

P = number of PERFORM exits and altered GO TO statements

C = any storage over 64K assigned to the program

It should be noted that the number of gn's is reduced when using OPT.

Within this configuration, assuming no Report Section, the compiler will accept for example:

300 procedure references assuming an average procedure-name length of 12 characters

25 OCCURS clauses with the DEPENDING ON option

10 files, assuming an average of 3 subordinate record entries

Effective Storage Considerations

The performance of the compiler is affected by the amount of storage it is allocated. The compiler will take advantage of any extra storage it is assigned. Furthermore, the use of a BUF parameter tailored to the work file device type in use is recommended. The following CBL parameters positively affect compile-time performance:

OPT
SYNTAX (CSYNTAX)
NOLIB
BUF

The amount of virtual storage within the compiler's partition and the limitation on the size of an individual internal table are two factors that limit the capacity of the compiler. The limitation on the size of internal tables can, in some instances, be overcome by the spilling over of some tables onto external devices. However, spilling over may cause a severe degradation of performance. The storage limitation should not be reached by any reasonable use of the language. However, within a limited storage capacity, excessive use of certain features and combination of features in the language could make compilation impossible. Some of the features that significantly affect storage usage are:

1. ADCON Table

   Each entry occupies 8 bytes. This table is not limited to the maximum size of 32,767 bytes. Entries are based on:

   - Number of 4096-byte segments in the Working-Storage Section
   - Number of 4096-byte segments in a file buffer area
   - Number of referenced procedure-names
   - Number of implicit procedure-name references such as those generated by IF, SEARCH, and GENERATE statements, ON SIZE ERROR, INVALID KEY, and AT END options, the OCCURS clause with the DEPENDING ON option, USE sentences, and the Segmentation Feature
   - Number of files

The size of this table is significantly reduced when using OPT.

2. Procedure-Name Table

   This table contains the number of definitions written in a section and unresolved procedure references. Procedure references are resolved at the end of a section if the definition of the procedure-name is in that section or a preceding section. Therefore, forward references beyond a section impact space.

3. OCCURS DEPENDING ON Table

   This table contains an entry for each unique object of an OCCURS clause with the DEPENDING ON option. The size of an entry is (2 + length of name + length of each qualifier) bytes.

4. Index Table

   An entry is made for each INDEXED BY clause consisting of 11 bytes for each index.

5. File Table

   An entry is made for each file specified in the program. Each entry occupies 60 bytes of storage.

6. Report Writer Tables

   A considerable amount of information is maintained concerning each RD such as controls, sums, headings, footings, routines to be generated, etc. The contents of the table is increased by the existence of qualification and subscripting in the Report Section. Approximately 30 reports can be processed, without exceeding the limit of a table.

7. Operand Table

   Entries are made depending on the number of operands in a statement. This table could reach its limits by the use of compound nested IF statements or GO TO DEPENDING ON statements with an excessive number of branch points.

8. Dictionary Table

   An entry is made for each procedure-name and each data-name in the program. A procedure entry consists of (7 or 9 + length of name) bytes. A data entry consists of (length of name + n) bytes, where n is determined by the attributes of the data item. Some of the features that contribute to the value n are:

   - One byte for each character in a numeric edited or alphanumeric edited item PICTURE clause.
   - Five bytes for an elementary item with a sterling report PICTURE clause.
   - Three bytes for an item subordinate to an OCCURS clause.

   In the statistics output, an indication is given if spill of this table occurred. If spill occurred, increasing the partition size assigned to the compiler should increase performance.

9. Literal Tables

   The total length of all literals (after optimization) may not exceed 32511 bytes. No more than 16255 literals may be specified.

   If the segmentation feature is used, an area corresponding to the total length of all optimized literals must be kept free during the time the ADCON table is being built. Therefore, a segmented program with literals may need more storage.

10. Miscellaneous Tables

   The existence of the following items causes entries to be made into tables and impacts the total space required for compilation.

   - SAME (RECORD) AREA clause
   - Subscripting
   - Intermediate Arithmetic Results
   - Complex Arithmetic Expressions
   - Complex Logical Expressions
   - APPLY clauses
   - Special-Names
   - RERUN clauses
   - Error messages
   - XREF
   - Segmentation feature
   - VERBSUM/VERBREF

EXECUTION TIME CONSIDERATIONS

The amount of virtual storage must be sufficient to accomodate at least:

- The selected control program
- Support for the file processing techniques used

- Load module to be executed

- Dynamic storage for VSAM, 3886 processing, and COUNT.

When the OPTIMIZE option is specified, the number of procedure blocks in the program cannot exceed 255. A procedure block is approximately 4096 bytes of Procedure Division code.

COBOL programs compiled with any of the symbolic debugging options (STATE, FLOW, SYMDMP) have different requirements at execution time than similar programs compiled without these options. The following differences should be noted:

- If the SYMDMP option is in effect, the work file required at compile time (SYS005) must be present at execution time.

- The size of the load module will increase by about 3200 bytes if the SYMDMP option is in effect. In addition, since the object-time subroutine that provides SYMDMP output is invoked dynamically, the programmer must provide space in the partition amounting to S + V. When only an abnormal termination dump is required, S = 4000 and V = 0; that is, 4000 extra bytes must be available. When dynamic dumps are required, S = 11,000 and V is approximately 25 * number of line-control cards + 10 * the number of identifiers specified on these line-control cards.

- The size of the load module will increase by 4500 + V bytes if the FLOW option is in effect. V is a variable factor that depends upon the number specified by the programmer on the CBL card. V is calculated using the formula:

  V = 92 + 4 * nn + 8 * p

  where "nn" is any number from 0 through 99, and "p" is the number of procedure-names in the program.

- The size of the load module will increase by 4600 + V bytes when the STATE option is in effect. V is approximately 5 * the number of COBOL statements in the program.

- When both SYMDMP and FLOW are in effect, the size of the load module will increase by the amount it would for FLOW alone, and the size of the partition increases by the amount it would for SYMDMP alone.

- A SIZE parameter must be specified on the EXEC card for VSAM and 3886 processing and if COUNT is requested on the CBL card.

COBOL programs with the execution frequency option COUNT have the following additional requirements:

- The size of the load module will increase by about 6000+V bytes (if any of the symbolic debugging options are in effect) to 8900+V bytes (if the symbolic debugging options are not in effect). V is calculated using the formula:

  V=(54 * pgm)+(8*nvb)+(7*npr)+((4+sym) * vbl)+pnl

  where

  pgm is the number of COPOL program units being monitored by COUNT

  nvb is the number of verbs in the program units

  npr is the number of procedure-names plus inserted procedure-names in the program

  sym is zero unless SYMDMP is in effect, then it becomes two

  vbl is the number of verb blocks in the program (which can be estimated as 1/3 the number of verbs in the program)

  pnl is the sum of the lengths of the procedure-names.

- The increase in dynamic storage in estimated using the formula

  D = 512+(72*pgm)+(4 * vbl)

  where

  pgm is the number of COBOL programs being monitored by COUNT

  vbl is the number of verb blocks in the program (which can be estimated as 1/3 the number of verbs in the program).

MULTIPROGRAMMING CONSIDERATIONS

In a system which supports the batch-job foreground (NPARTS = 2 or more) and private core-image library options, the Linkage Editor can execute in any foreground partition (as well as the background partition) provided a minimum of 14K or 64K of storage is assigned to the partition. When executing in a foreground partition, a private core image library must be assigned.

In the multiprogramming environment described above, the COBOL compiler can be executed in any partition having a minimum of 64 bytes in the following manner:

> At system generation time, link edit the compiler in the background partition and place it in the system core image library.

SORT FEATURE CONSIDERATIONS

The DOS/VS SORT/MERGE Program Product, Program Number 5746-SM1, must be executed under control of DOS/VS. It requires the following minimum machine configuration:

1. The DOS/VS SORT/MERGE Program Product uses 16K bytes; additional storage is needed for DOS/VS and for user-written routines (that is, the COBOL program, etc.).

   Note: Performance often increases significantly if 50K is available for operation of the Sort/Merge program. At the 100K level, the performance could be even higher.

2. Standard instruction set.

3. At least one 2314, 2319, 3330, 3333, 3340, or 3350 work file. (System residence requirements may necessitate having an additional disk storage unit for sorting.)

4. One IBM 1403, 1443, or 3211 Printer, or one IBM operator communication device (for example, 3215).

5. One IBM 1442, 2501, 2520, 2540, 3505, 3525, or 2560 Card Reader, or one IBM 2400 or 3400 Series Magnetic Tape Unit (7- or 9-track) assigned to SYSIPT and SYSRDR.

6. Three IBM 2400 or 3400 Series Magnetic Tape Units for work files when tape units are to be used for intermediate storage.

For specific size, device, and work file requirements of the other Sort/Merge products, see the respective Programmer's Guides as noted in the preface.

Note: If a size parameter is used in the //EXEC statement, it should be used as follows:

```
//EXEC,SIZE=(AUTO,nK)
```

where NK has to meet specific Sort storage requirements.

PROGRAM COMMUNICATION

For each partition, the supervisor contains a storage area called the communication region. The supervisor uses the communication region, and your program also can use it. Your program can check the communication region of the partition in which your program runs; your program can also modify the user area of this communication region.

Figure 68 shows the portion of the communication region containing information of interest. This information is also described below.

Byte(s)	Information
0-7	Calendar date. Supplied from system date whenever the JOB statement is encountered. The field can be two forms: mm/dd/yy or dd/mm/yy where mm is month, dd is day, yy is year. It can be temporarily overridden by a DATE statement.
8,9	Address of the problem program area.
10,11	Address of the beginning of the problem program area.
12-22	User area for communication within a job step or between job steps. All 11 bytes are set to zero when the '//JOB' JOB control statement is encountered. Note: The COBOL compiler uses bytes 12 and 13.
23	UPSI (user program switch indicators). Set to binary zero when the JOB statement for the job is encountered. Initialized by UPSI job control statement. The condition-name associated with the status of the UPSI switches can be specified in the COBOL program via the Special-Names paragraph of the Environment Division. The condition-name associated with each may be tested in the Procedure Division of the COBOL program.
24-31	Job name as found in the JOB statement for the job.
32-35	Address of the uppermost byte of the program area. If the program was initiated with the SIZE parameter in the EXEC job control statement, this address gives the highest duty of the area determined by the SIZE parameter. If the SIZE parameter was not specified, the address is the highest address in the partition (either real or virtual).
36-39	Address of the uppermost byte of the current phase placed in the program area by the last FETCH or LOAD macro in the job.

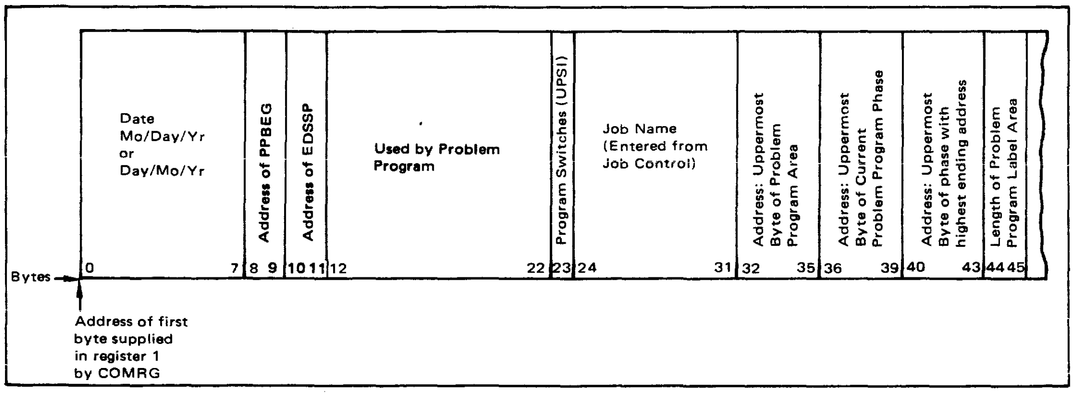

Date Mo/Day/Yr or Day/Mo/Yr
Address of PPBEG
Address of EDSSP
Used by Problem Program
Program Switches (UPSI)
Job Name (Entered from Job Control)
Address: Uppermost Byte of Problem Program Area
Address: Uppermost Byte of Current Problem Program Phase
Address: Uppermost Byte of phase with highest ending address
Length of Problem Program Label Area
Bytes
0 7 8 9 10 11 12 22 23 24 31 32 35 36 39 40 43 44 45
Address of first byte supplied in register 1 by COMRG

Figure 68. Communication Region in the Supervisor

Byte(s)	Information
40-43	Highest ending virtual storage address of the phase among all the phases having the same first four characters as the operand on the EXEC statement. For the background partition only, job control builds a phase directory of these phases. The address may be incorrect if the program loads any of these phases above its link-edited origin address and the relocating loader is not used. If the EXEC statement has no operand, job control places in this location the ending address of the phase just link-edited.
44,45	Length of program label area.

The COM-REG special register may be used to access bytes 12-22 of the communication region.

This appendix illustrates the necessary job control statements and their sequence for five typical programs:

1. Creating a Direct File
2. Retrieving and Updating a Direct File
3. Creating an Indexed File
4. Retrieving and Updating an Indexed File
5. Sorting an Unlabeled Tape File

In all five programs the programmer has requested the following compiler options through the OPTION control statement:

NODECK	-- No punched card output for the object program is needed.
LINK	-- The object module is to be linkage edited.
LIST	-- The COBOL source statements are to be printed on SYSLST.
LISTX	-- A Procedure Division map with global tables, literal pool, and register assignments is to be printed on SYSLST.
SYM	-- A Data Division map is to be printed on SYSLST.
ERRS	-- The diagnostic messages of the COBOL compiler are to be printed on SYSLST.

The EXEC FCOBOL statement calls for execution of the FCOBOL compiler.

By using the CBL card, the programmer indicates that in this source program the quotation mark (") is used for nonnumeric literals.

The ASSIGN clause in the COBOL source program specifies a system-name with the following fields:

```
(Non-VSAM)
    SYSnnn-class-device-organization[-name]

(VSAM)
    SYSnnn[-class][-device][-organization]
      [-name]
```

The ASSGN control statement for a file must specify the same logical unit as the <u>SYSnnn</u> field of system-name. The ASSGN statement assigns the logical unit to a specific hexadecimal address. The address specified must be associated with the device whose number is given in the <u>device</u> field of system-name.

The DLBL control statement for a labeled file on a mass storage device must contain the same <u>name</u> as system-name. This is the name by which the file is known to the control program. (The <u>name</u> field of system-name is optional. If <u>name</u> is omitted, the DLBL statement must specify the logical unit (SYSnnn) as the file-name.) The code field of the DLBL statement must correspond to the <u>class</u> and <u>organization</u> fields of system-name as follows:

DLBL "code"	ASSIGN "class"	ASSIGN "organization"
SD	DA or UT	S
		AS (entry-sequenced file) omitted (key-sequenced file)
DA	DA	A or U, D or W
ISC	DA	I
ISE	DA	I

If SYSnnn is omitted from the first EXTENT control statement for a file on a mass storage device, then the logical unit is determined from the SYSnnn field of the COBOL system-name; if SYSnnn is included in the first EXTENT statement and differs from SYSnnn of the system-name, the EXTENT card specification overrides the COBOL source specification. (Subsequent EXTENT statements for the same file, if they immediately follow the first, may omit this field.) The type of the extent must be compatible with the <u>organization</u> field of system-name as follows:

EXTENT "type"		ASSIGN "organization"
1	(data area, no split cylinder)	S, A, U, I, D, W
		AS
2	(overflow area for indexed file)	I
4	(index area for indexed file)	I
8	(data area, split cylinder)	S, A, U, I, D, W

DIRECT FILES

The following two examples illustrate the job control statements necessary for programs that create and update a direct file.

In the COBOL source programs, the programmer has written:

```
    SELECT DA-FILE ASSIGN TO
      SYS015-DA-2311-A-MASTER...

    SELECT CARD-FILE ASSIGN TO
      SYS007-UR-2540R-S...
```

In the READFILE source program, the programmer has written:

```
    SELECT PRINT-FILE ASSIGN TO
      SYS008-UR-2403-S...
```

(Note the relationship between the system-names in the source programs and the control statements.)

The LBLTYP statement defines the amount of storage to be reserved to process labels for the DA file. The file has one extent.

The EXEC LNKEDT statement causes the object program to be link edited.

An ASSGN control statement assigns logical unit SYS007 to the hexadecimal address 00C -- a 2540R Card Reader.

In the updating program, another ASSGN statement assigns logical unit SYS008 to the hexadecimal address 00E -- a 1403 Printer.

The next series of statements identify the direct file completely.

The ASSGN statement identifies the file as residing on logical unit SYS015, which has the hexadecimal address of 192 -- a 2311 Disk Drive.

The DLBL statement specifies the filename as MASTER, with an expiration date of the 365th day of 1973, and that the file has direct organization (DA).

The EXTENT statement specifies that the file residing on logical unit SYS015 has a serial number 111111, that the extent is a data area with no split cylinder and that this is the first (and only) extent for the file (type and sequence number 1,0), that the file begins on relative track 1020 (track 0 of cylinder 102), and that the file occupies 100 tracks.

(Note that in the EXTENT statement, the relative track number (1020) is not required for the input DA file of the updating program, since the system will use the file labels for this information.)

The EXEC statement begins execution of the problem program, and is followed by input data.

The /* statements indicate end-of-data, the /& statement indicates end-of-job.

Creating a Direct File

```
// JOB CREATEDA
// OPTION NODECK,LINK,LIST,LISTX,SYM,ERRS
// EXEC FCOBOL
 CBL QUOTE

   {COBOL source deck}
/*
// LBLTYP NSD(01)
// EXEC LNKEDT
// ASSGN SYS007,X'00C'
// ASSGN SYS015,X'192'
// DLBL MASTER,,74/365,DA
// EXTENT SYS015,111111,1,0,1020,100
// EXEC

   {input data cards}
/*
/&
```

Retrieving and Updating a Direct File

```
// JOB READFILE
// OPTION NODECK,LINK,LIST,LISTX,SYM,ERRS
// EXEC FCOBOL
 CBL QUOTE

   {COBOL source deck}
/*
// LBLTYP NSD(01)
// EXEC LNKEDT
// ASSGN SYS007,X'00C'
// ASSGN SYS008,X'00E'
// ASSGN SYS015,X'192'
// DLBL MASTER,,74/365,DA
// EXTENT SYS015,111111,1,0,1020,100
// EXEC

   {input data cards}
/*
/&
```

INDEXED FILES

The following two examples illustrate the job control statements necessary for programs that create and update an indexed file.

In the CREATEIS source program, the programmer has written:

```
    SELECT IS-FILE ASSIGN TO
      SYS015-DA-2314-I-MASTER
    ACCESS IS SEQUENTIAL
    RECORD KEY IS REC-ID.
```

In the RANDIS source program, the programmer has written:

```
    SELECT IS-FILE ASSIGN TO
      SYS015-DA-2314-I-MASTER
    ACCESS IS RANDOM
    NOMINAL KEY IS KEY-ID
    RECORD KEY IS REC-ID.

    SELECT PRINT-FILE ASSIGN TO
      SYS008-UR-1403-S
    RESERVE NO ALTERNATE AREAS.
```

In both source programs, he has written:

```
    SELECT CARD-FILE ASSIGN TO
      SYS007-UR-2540R-S.

I-O-CONTROL.
    APPLY MASTER-INDEX TO 2311 ON IS-FILE.
```

(Note the relationship between the source program statements and the job control statements.)

The LBLTYP statement defines the amount of storage reserved to process labels for the indexed file. The file has three extents: a master index extent, a cylinder index extent, and a data extent.

The EXEC LNKEDT statement causes the object module to be link edited.

An ASSGN control statement assigns logical unit SYS007 to the hexadecimal address 00C -- a 2540R Card Reader.

In the retrieval program, another ASSGN statement assigns logical unit SYS008 to the hexadecimal address 00E -- a 1403 Printer.

The next ASSGN statement assigns logical unit SYS015 to the hexadecimal address 193 -- a 2314 Disk Drive.

The DLBL statement names the file as MASTER, and indicates the expiration date as the 365th day of 1974. In the file creation program, the file label is indexed sequential using Load Create (code ISC); in the retrieval program, the file label is indexed sequential using Load Extension, Add or Retrieve (code ISE).

The first EXTENT statement is identified as a master index (type and sequence numbers are 4,0), and the relative track is | 900 (the extent begins on cylinder 90 track 0), and the extent is 20 tracks long.

The second EXTENT statement is identified as a cylinder index (type and sequence number are 4,1), the relative track is 1820 (the extent begins on cylinder 91, track 0), and the extent is 20 tracks long.

(Note that the extents assigned to master and cylinder indexes must be contiguous, and that the master index must precede the cylinder index on the disk pack. Also note, that if a master index is not requested, the first extent is that for the cylinder index, which would be type 4, sequence number 1.)

The third EXTENT statement is identified as a data area (type 1) and is the third extent named for this file. The relative track is 0020 (the extent begins on cylinder 1, track 0), and the extent is 1760 tracks long.

End-of-data is indicated with the /* statement; end-of-job is indicated with the /& statement.

<u>Creating an Indexed File</u>

```
// JOB CREATEIS
// OPTION NODECK,LINK,LIST,LISTX,SYM,ERRS
// EXEC FCOBOL
 CBL QUOTE

   {COBOL source deck}
/*
// LBLTYP NSD(03)
// EXEC LNKEDT
// ASSGN SYS007,X'00C'
// ASSGN SYS015,X'193'
// DLBL MASTER,,74/365,ISC
// EXTENT SYS015,111111,4,0,1800,20
// EXTENT SYS015,111111,4,1,1820,20
// EXTENT SYS015,111111,1,2,0020,1760
// EXEC

   {input data card}
/*
/&
```

Retrieving and Updating an Indexed File

```
// JOB RANDIS
// OPTION NODECK,LINK,LIST,LISTX,SYM,ERRS
// EXEC FCOBOL

   {COBOL source deck}
// LBLTYP NSD(03)
// EXEC LNKEDT
// ASSGN SYS007,X'00C'
// ASSGN SYS008,X'00E'
// ASSGN SYS015,X'193'
// DLBL MASTER,,73/365,ISE
// EXTENT SYS015,111111,4,0,1800,20
// EXTENT SYS015,111111,4,1,1820,20
// EXTENT SYS015,111111,1,2,0020,1760
// EXEC

   {input data cards}
/*
/&
```

FILES USED IN A SORT OPERATION

The following example illustrates the job control statements necessary for a program that sorts an unlabeled tape file.

In the COBOL source program, the programmer has written:

```
    SELECT NET-FILE-IN ASSIGN TO
      SYS007-UT-2400-S.

    SELECT NET-FILE-OUT ASSIGN TO
      SYS008-UT-2400-S.

    SELECT NET-FILE ASSIGN TO 3
      SYS001-UT-2400-S.
```

NET-FILE-IN is the input file; NET-FILE-OUT is the output file; NET-FILE is the sort work file, which utilizes three tape units.

(Note the relationship between the system-names in the COBOL source program and the control statements.)

The EXEC LNKEDT statement causes the job to be link edited.

The first two ASSGN control statements assign the logical unit SYS007 to hexadecimal address 181, and logical unit SYS008 to hexadecimal address 182. SYS007 is the sort input file, and SYS008 is the sort output file.

The last three ASSGN statements assign logical unit SYS001 to hexadecimal address 183, logical unit SYS002 to hexadecimal address 281, and logical unit SYS003 to hexadecimal address 282. SYS001, SYS002, and SYS003 are the logical units that must be used for sort work files. The sort work files must be assigned to 9-track tape units. At this installation, 9-track tape drives are associated with hexadecimal addresses 183, 281, and 282.

Sorting an Unlabeled Tape File

```
// JOB SORTCOB
// OPTION NODECK,LINK,LIST,LISTX,SYM,ERRS
// EXEC FCOBOL
 CBL QUOTE

   {COBOL source deck}
// EXEC LNKEDT
// ASSGN SYS007,X'181'
// ASSGN SYS008,X'182'
// ASSGN SYS001,X'183'
// ASSGN SYS002,X'281'
// ASSGN SYS003,X'282'
// EXEC
/&
```

This appendix contains information on how to generate a listing of compile-time diagnostic messages.

COMPILE-TIME MESSAGES

The user can request a complete listing of the diagnostics generated by this compiler simply by compiling a program with a PROGRAM-ID of ERRMSG. For a description of the formats of compiler diagnostics and information about generating this listing, see the chapter entitled "Output" in this publication.

OPERATOR MESSAGES

This section lists the messages issued to SYSLOG by the IBM DOS/VS COBOL Compiler and Library. All of the messages listed are also issued on SYSLST.

The following messages are issued during compilation on SYSLOG. They are also printed on SYSLST with the prefix ILA.

| C100I PARTITION IS LESS THAN 64K

| Explanation: At least 64K is required to compile using DOS/VS COBOL. Probable user error.

System Action: The compilation is terminated.

Programmer Response: Not applicable.

Operator Response: Use the ALLOC command to allocate at least 64K to the partition (refer to BUF option). If the problem recurs, do the following to complete your problem determination action before calling IBM for programming support.

1. Execute the MAP command and save the output.
2. Have the source deck, control cards, output listing, and console sheet available.

C101I DEVICE NOT ASSIGNED - SYSnnn.

Explanation: (nnn is either 001, 002, 003, or 004.) The specified logical unit is unassigned and must be assigned. Probable user error.

System Action: The compilation is terminated.

Programmer Response: Not applicable.

Operator Response: Use the ASSGN command to assign a physical unit (magnetic tape or disk) to the file indicated. If the problem recurs, do the following to complete your problem determination action before calling IBM for programming support:

1. Execute the LISTIO command and save the output.

2. Have the source deck, control cards, output listing, and console sheet available.

C102I UNSUPPORTED DEVICE TYPE - SYSnnn.

<u>Explanation</u>: (nnn is either 001, 002, 003, or 004.) The specified file must be a tape or disk file for SYS002 through SYS004. SYS001 should be assigned to disk; however, in small, simple programs that do not require dictionary spill, it is sometimes possible to compile with the spill file (SYS001) assigned to tape. If any spill does occur, an input/output error may occur. Compile-time statistics will say "DICTIONARY SPILL HAS OCCURRED". No mention is made of dictionary spill in the compile-time statistics if spill does not occur. Probable user error.

<u>System Action</u>: The compilation is terminated.

<u>Programmer Response</u>: Not applicable.

<u>Operator Response</u>: Use the ASSGN command to assign the appropriate physical unit to the file indicated -- SYS001 should be assigned to a magnetic tape or disk unit. If the problem recurs, do the following to complete your problem determination action before calling IBM for programming support:

1. Execute the LISTIO command and save the output.

2. Have the source deck, control cards, output listing, and console sheet available.

C103I END OF FILE ON SYSIPT.

<u>Explanation</u>: End-of-file was encountered in the initialization phase; no source statements were found. Probable user error.

<u>System Action</u>: The compilation is terminated.

<u>Programmer Response</u>: Not applicable.

<u>Operator Response</u>: Ensure that a /* card does not precede the source deck, or add the source deck to the job stream. If the problem recurs, do the following to complete your problem determination action before calling IBM for programming support:

1. Execute the LISTIO command and save the output.

2. Have the source deck, control cards, output listing, and console sheet available.

C104I SYS001 FILE NOT ASSIGNED TO DISK

<u>Explanation</u>: In small, simple programs that do not require dictionary spill, it is sometimes possible to compile with the spill file (SYS001) assigned to tape. However, if any spill does occur, an input/output error may occur. Any compilation which spills the dictionary will contain a message in compile-time statistics. User error.

<u>System Action</u>: The compilation continues.

<u>Programmer Action</u>: Not applicable.

Operator Response: Use the ASSGN command to assign SYS001 to a disk unit. If the problem recurs, do the following to complete your problem determination action before calling IBM for programming support:

1. Execute the LISTIO command and save the output.

2. Have the source deck, control cards, output listing, and console sheet available.

C105I W-CANNOT OPEN SYS005 -- SYMDMP IGNORED.

Explanation: The SYMDMP option has been specified, but the file needed for symbolic debug cannot be opened since SYS005 is unassigned. Probable user error.

System Action: The SYMDMP option is canceled, and the compilation continues.

Programmer Response: Not applicable.

Operator Response: Use the ASSGN command to assign SYS005 to a physical unit. If the problem recurs, do the following to complete your problem determination before calling IBM for programming support:

1. Execute the LISTIO command and save the output.

2. Have the source deck, control cards, output listing, and console sheet available.

C106I SYS006 IS NOT A DISK. NOLVL ASSUMED.

Explanation: The specified logical unit is not assigned to a disk.

System Action: Compilation continues with NOLVL.

Programmer Response: Not applicable.

Operator Response: Use the ASSGN command to assign SYS006 to a disk unit.

OBJECT-TIME MESSAGES

The following messages are normally issued on SYSLOG.

C110A STOP literal

Explanation: The programmer has issued a STOP literal statement in the COBOL source program.

System Action: Awaits operator response.

Programmer Response: Not applicable.

Operator Response: Operator should respond with end-of-block, or with any character in order to proceed with the program.

C111A AWAITING REPLY

Explanation: This message is issued in connection with the Full American National Standard COBOL ACCEPT statement.

System Action: Awaits operator response.

Programmer Response: Provide the operator with instructions.

Operator Response: The operator should reply as specified by the programmer.

The following messages are issued on SYSLOG and SYSLST prior to cancellation of the job. If the DUMP option is specified, a partial dump is taken from the problem program origin to the highest storage location of the last phase loaded. When this occurs, the eight bytes immediately preceding the DTF are destroyed. The messages have the form:

CmmmI SYSnnn filename DTFaddress text

where:

nnn is equal to 001 through 255
filename is seven or fewer characters and is generated from the file-name specified in the SELECT sentence.
address is the hexadecimal address of the file's DTF table.
mmm and text correspond as follows:

mmm	text
112	DATA CHECK
113	WRONG LENGTH RECORD
114	PRIME DATA AREA FULL
115	CYLINDER INDEX TOO SMALL
116	MASTER INDEX TOO SMALL
117	OVERFLOW AREA FULL
118	DATA CHECK IN COUNT
119	DATA CHECK IN KEY OR DATA
120	NO ROOM FOUND
121	DASD ERROR
122	DASD ERROR WHILE ATTEMPTING TO WRITE RECORD ZERO
123	FILE CANNOT BE OPENED AFTER CLOSE WITH LOCK
124	CYLINDER AND MASTER INDEX TOO SMALL
125	NO EXTENTS
127	NO EOF RECORD WRITTEN IN PRIME DATA AREA
128	UNRECOVERABLE I/O ERROR
129	3540 EQUIPMENT CHECK
130	INPUT/OUTPUT ERROR. FILE STATUS SET TO XX NEAR REL LOC. XXXXXX.
131	USABLE TO OPEN FILE 'SYSnnn'. CANCELING.
132	SIZE NOT SPECIFIED OR INSUFFICIENT GETVIS AREA.
140	INVALID SEPARATE SIGN CONFIGURATION.

Explanation: Condition indicated occurred on SYSnnn.

System Action: The job is cancelled.

Programmer Response: Rerun the job or add a user Declaractive Section to the Procedure Division of the source program to handle errors within the program.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, compiler output, and console sheet available.

Operator Response: Not applicable.

C125I NO EXTENTS

Explanation: During CLOSE UNIT processing, no extent is found for the next volume.

System Response: The job is cancelled.

Programmer Response: Rerun job with proper EXTENT (XTENT) statements.

Operator Response: Not applicable.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, compiler output, and console sheet available.

The following message is issued on SYSLOG:

C126D SYSnnn IS IT EOF?

Where nnn is equal to 001 through 255

Explanation: A tapemark was just read on an unlabeled tape file described at compilation time as having more than one reel.

System Action: Awaits response from operator.

Programmer Response: Not applicable.

Operator Response: The operator must respond either with N if is end of volume, or with Y if it is end of file.

The following messages are issued on SYSLOG and SYSLST:

C127D NO EOF RECORD WRITTEN IN PRIME DATA AREA

Explanation: During CLOSE processing of an ISAM file opened OUTPUT, no room was found to write EOF record.

Programmer Response: Rerun the job with the proper EXTENT.

If the problem recurs, do the following to complete your problem determination action before calling IBM for programming support. Have source deck, control cards, compiler output, and console sheet available.

Operator Response: Not applicable.

C128D UNRECOVERABLE I/O ERROR

Explanation: This is probably a hardware error on tape.

Programmer Response: Not applicable.

Operator Response: Rerun the job.

If the problem recurs, do the following to complete your problem determination action before calling IBM for programming support. Have source deck, control cards, compiler output, and console sheet available.

C129I VSAM SUBROUTINE ERROR. CANCELING JOB.

Explanation: The subroutine has encountered an unrecoverable error. This can occur when a VSAM OPEN, CLOSE, or ACTION request (GET, PUT, etc.) returns an error code from which the subroutine has no means of recovering, or when one of the VSAM macros (SHOWCB, GENCB, etc.) returns a non-zero return code. All such conditions indicate an error found in the subroutines and/or in VSAM.

Action: The program is canceled with a dump.

Programmer Response: Submit an APAR with the dump.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C130I INPUT/OUTPUT ERROR. FILE STATUS SET TO xx NEAR REL LOC. xxxxxx.

Explanation: An I/O error has occurred on the file being accessed by the COBOL statement at or near the relative location given in the message, and the user has no USE declarative for that file.

Action: Control returns to COBOL at the statement following the COBOL request that caused the error.

Programmer Response: If the error occurred on a READ operation, processing can continue. If the error occurred on a WRITE operation, there may be a loss of data.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C131I UNABLE TO OPEN FILE 'SYSnnn'. CANCELING.

Explanation: The VSAM OPEN or CLOSE request gave a return code of X'68' or X'6C' because of invalid time stamps in the VSAM catalog or VSAM file. The VSAM catalog or file should be recreated. See DOS/VS Supervisor and I/O Macros for more detail on the OPEN/CLOSE return codes.

Action: The job is canceled.

Programmer Response: Recreate the VSAM catalog and/or file.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C132I SIZE NOT SPECIFIED OR INSUFFICIENT GETVIS AREA.

Explanation: A GETVIS SVC to obtain GETVIS space for VSAM control blocks was unsuccessful.

Action: The job is canceled.

Programmer Response: Increase the partition size and resubmit the job with a SIZE parameter in the EXEC statement.

C140I INVALID SEPARATE SIGN CONFIGURATION

Explanation: During execution of a COBOL program, an invalid sign was detected for a separately signed item.

Action: The job is terminated.

Programmer Response: Probable user error. Correct program's input data before reexecuting.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, compiler output, and data available.

The following messages (C150I-C170I) are listed on SYSLST. The messages have the form:

$$\text{CmmmI} \begin{Bmatrix} \text{program-id} \\ \\ \text{card/verb number} \end{Bmatrix} \quad \text{text}$$

Messages C150I through C162I may appear interspersed among the SYMDMP control cards at the point at which the error is encountered. Program-id is provided for all messages except C150I through C152I. For these, the card/verb number of the corresponding line-control card is given instead. The program-id associated with C150I through C152I can be determined from the nearest preceding program-control card.

Messages C153I through C155I may also appear in the midst of the dump output if the error condition is not recognized until dumping has started.

C150I IDENTIFIER NOT FOUND.

Explanation: An identifier specified on the line-control card cannot be found in the program or is invalid. Level-66 and

level-88 items and items defined under an RD are invalid requests.

Action: The dump request for this identifier is ignored.

Programmer Response: Probable user error. Before reexecuting, ensure that no requests have been made on the line-control card for the dumping of identifiers that have not been defined or that are invalid.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C151I CARD NUMBER NOT FOUND

Explanation: The card number specified on the line-control card is not within range of the Procedure Division.

Action: The line-control card which specifies the nonexistent card number is skipped.

Programmer Response: Probable user error. Ensure that any card number specified on a line-control card is within range of numbers specified for source program before reexecuting.

If the problem recurs. do the following before calling IBM for programming support: have source deck, control cards, and compiler output abailable.

C152I VERB NUMBER NOT FOUND

Explanation: The verb number specified on a line-control card does not exist on the card specified.

Action: The nearest verb number on the card specified is used.

Programmer Response: Probable user error. Correct verb number specification before reexecuting.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C153I NO ROOM TO DUMP.

Explanation: If this message immediately follows a program-control card, sufficient storage is not available for the debug subroutine or for the 72 bytes of data required for each program in the run unit. If this message follows an abnormal termination message, one or more of the following is not available in free storage or in the COBOL Procedure Division: a contiguous block of 4000 bytes, a contiguous block of 1800 bytes, or a contiguous block of 512 bytes.

Action: No Data Division dump for the indicated program and, in some instances, no statement number information, is provided.

Programmer Response: Probable user error. Increase the size of the partition before reexecuting. See "System Configuration" for information about storage requirements for symbolic debugging.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C154I I/O ERROR ON DEBUG FILE.

Explanation: An input/output error has occurred on the debug file. Note that such an error may be the result of a file other than the debug file being mounted on the logical unit specified.

Action: SYMDMP output is cancelled for the indicated program.

Response: Hardware, operator, or user JCL error. Before reexecuting, check logical unit number specified on program-control card against current mounting, as well as the ASSGN, DLBL, and EXTENT cards of compilation.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C155I WRONG DEBUG FILE FOR PROGRAM.

Explanation: The file corresponding to the filename and/or logical unit number provided on the program-control card is not the debug file created for this program at compile time.

Action: SYMDMP output is cancelled for the indicated program.

Programmer Response: Probable user error. Before reexecuting, ensure that the filename and/or logical unit specified on the program-control card corresponds to that of the debug file created at compile time.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C156I NO ROOM FOR DYNAMIC DUMP.

Explanation: Sufficient storage is not available to store the line-control card information during execution.

Action: Dynamic dumping is cancelled for the indicated program.

Programmer Response: Probable user error. Increase size of partition or decrease number of line-control cards before reexecuting.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C157I INVALID FILENAME.

Explanation: If the "filename" parameter is specified for a disk file on the CBL card at compile time, the same "filename" must also be specified on the program-control card. "Filename" may be from one to seven characters in length; the first character must be a letter.

Action: All SYMDMP output is cancelled for the indicated program.

Programmer Response: Probable user error. Correct "filename" specification on the program-control card before reexecuting.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C158I INVALID LOGICAL UNIT.

Explanation: The logical unit parameter on the program-control card must be specified, must be an integer between 0 and 244, and must match the one specified in the ASSGN control statement for the debug at compile time.

Action: All SYMDMP output is cancelled for the indicated program.

Programmer Response: Probable user error. Correct logical unit specification on program-control card before reexecuting.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C159I MISSING PARAMETERS.

Explanation: A non-continued line-control card ends with (HEX), OF, IN, or THRU. Possibly a continuation punch is missing in column 72.

Action: A HEX or THRU option ending a card is ignored. When a card ends with OF or IN, the word is ignored and the identifier that is dumped is the first one encountered whose qualifiers match those preceding the word OF or IN.

Programmer Response: Probable user error. Check line-control card for keypunch errors before reexecuting.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C160I INVALID OPTION.

Explanation: An element used as an optional parameter on a program-control card is not one of the legal program-control card options.

Action: The element is ignored.

Programmer Response: Probable user error. Correct syntax of program-control card before reexecuting.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C161I SUBSCRIPTING ILLEGAL.

Explanation: The "name" parameter of the line-control card may not be subscripted.

Action: The subscripts are ignored. Every occurrence of the identifier is dumped.

Programmer Response: Probable user error. Specify the name of the item without the subscript before reexecuting. This will result in a dump of every occurrence of the item.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C162I ON PARAMETER TOO BIG.

Explanation: Neither the n, m, nor k parameter of the ON option may exceed 32767.

Action: The number is reduced to 32767.

Programmer Response: Probable user error. Correct invalid parameter before reexecuting.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C163I FLOW TRACE NON-CONTIGUOUS. MORE THAN 10 PROGRAMS ENCOUNTERED

Explanation: A non-contiguous flow trace will result if FLOW option is effective in a subprogram structure of more than 10 programs compiled with the FLOW option.

Action: The FLOW trace is terminated upon encountering the eleventh PROGRAM-ID. Tracing resumes only upon returning to one of the original ten programs.

Programmer Response: Probable user error. If trace is absent for a program where it is critical, recompile one or more of the programs where the flow is non-critical without the FLOW option and reexecute.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C164I FLOW TRACE IN EFFECT BUT NO PROCEDURES TRACED.

Explanation: Abnormal termination has taken place before any COBOL statement with a procedure-name has been traced.

Action: No tracing is done.

Programmer Response: Probable user error. If trace is desired, recompile the program after inserting additional procedure-names.

If the problem recurs, do the following before calling IBM for progromming support: have source deck, control cards, and compiler output available.

C165I SYMDMP/STATE/FLOW/COUNT INTERNAL ERROR. EXECUTION CANCELLED.

Explanation: Abnormal termination occurred during execution of one of the debugging subroutines.

Action: The job is cancelled.

Programmer Response: Internal logic error.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C169I STATE OPTION CANCELLED.

Explanation: compiler or logic error has occurred during STATE option processing. Under certain conditions, this error may result from other user errors. For example, a loop might destroy some of the information required by the STATE subroutines; an invalid branch might cause a non-existent priority-number to be stored in the TGT, etc.

Action: STATE output is cancelled.

Programmer Response: Probable user error. Possible compiler error or user error. Correct other known errors (if any) before attempting reexecution.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C170I INVALID ADDRESS.

Explanation: The address calculated for a subscripted identifier, or a starting or ending address of a variable-length identifier used as the receiving field in a MOVE statement is invalid.

Action: A symbolic dump is produced.

Programmer Response: Probable user error. Possible compiler error or user error. Correct other known errors (if any) before attempting reexecution.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C171I SPACE NOT FOUND FOR THE COUNT CHAIN. CONTINUING.

Explanation: A GETVIS macro was unsuccessful due to lack of space.

Action: Execution continues. Execution statistics are not provided for the last indicated program unit.

Programmer Response: Probable user error. Allocate more space on EXEC card before attempting reexecution.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C172I SPACE NOT FOUND FOR THE VERBSUM TABLE. CONTINUING.

Explanation: A GETVIS macro was unsuccessful due to lack of space.

Action: Execution continues. Verb summary statistics are not provided for the program.

Programmer Response: Probable user error. Allocate more space on EXEC card before attempting reexecution.

C173I FREEVIS FAILED. EXECUTION CANCELLED.

Explanation: A FREEVIS macro was unsuccessful.

Action: Execution is terminated.

Programmer Response: Probable user error. Allocate more space on EXEC card before attempting reexecution.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C175I INVALID COUNT TABLE ENTRY. EXECUTION CANCELLED.

Explanation: A count table entry in the object module is not one of the following: end-of-table indicator, procedure-id, or verb-id.

Action: Execution is terminated.

Programmer Response: Probable user error. Possible compiler or user error. Check your program for routines that may have moved data into the count table area. Correct other known errors (if any) before attempting execution.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

COBOL OBJECT PROGRAM UNNUMBERED MESSAGES

xxx...

Explanation: This message is written on the console and is recognizable because it is not preceded by a message code and action indicator. It is issued by an object program originally coded in COBOL . The message text is supplied by the object program and may indicate alternative action to be taken.

System Action: The job continues.

Operator Response: Operator response, if any is needed, is determined by the message text.

This appendix contains information on the 3886 Optical Character Reader, Model 1* (denoted as "the OCR"). Topics discussed include:

3886 OCR processing
COBOL considerations for 3886 OCR processing.
Status key values
Sample program

This discussion assumes familiarity with these IBM 3886 Optical Character Reader publications:

IBM 3886 OCR General Information Manual, Order No. GA21-9146 -- for terminology, device capabilities, and the formats of the header and data records.

IBM 3886 OCR Input Document Design and Specifications, Order No. GA21-9148 -- for document design considerations and detailed specifications.

In addition, the applicable portions of the following manuals should be referenced:

IBM DOS/VS Supervisor and I/O Macros, Order No. GC33-5373 -- for describing documents using the DFR and DLINT macros.

IBM DOS/VS System Generation, Order No. GC33-5377

IBM DOS/VS Data Management Guide, Order No. GC24-5062.

IBM DOS/VS Program Planning Guide for the IBM 3886 Optical Character Reader, Model 1, Order No. GC21-5059

3886 OCR PROCESSING

The 3886 OCR, Model 1 is a general purpose online device that satisfies a broad range of data entry requirements. The OCR accepts documents sized from 3 inches by 3 inches to 9 inches by 12 inches. It can read machine-printed alphabetic characters, numeric characters, and certain special characters in a wide variety of fonts, as well as hand-printed numeric characters.

The OCR reads documents one line at a time, under program control. Additional features, all under program control, include:

- document marking
- line marking
- document eject (with stacker selection)
- line reread (for the current line, and with a different format if desired)

Note: The OCR cannot read previous lines; reading can proceed from top to bottom on the document only.

IMPLEMENTING AN OCR OPERATION

Document Design

The OCR form that will be used for input should be prepared independently of the COBOL program. Document design criteria are described in detail in IBM 3886 Optical Character Reader, Input Document Design Guide and Specifications.

The most important aspects of document design are:

1. The locations of lines which can be read. These lines are identified by "timing marks." Lines not associated with timing marks are always ignored by the OCR. Note that lines may be almost anywhere on the document, and need not be at regular intervals.

2. The location of fields to be read. Fields, (strings of related characters) should be identified in document design. They should be described using the DFR and DLINT macros. (See section entitled "Document Description".)

3. The form identifier. This field should be a pre-printed code at a common location on the first readable line of each format. This field can be ignored by programming or DLINIT

----
*This device should not be confused with the 3886, Model 2, an offline Optical Character Reader with output to tape. Information is included in this chapter on processing the tapes produced by the Model 2.

specification if desired; it should, however, be included in the form design so as to allow for later form changes or intermixing of forms in batches without disruption of operations.

Document Description

Documents are described in the system with the Define Format Record (DFR) and Define Line Type (DLINT) macros. These macros should be coded independently of the COBOL program.

The DFR macro identifies, by name, a collection of DLINT macros, and establishes various default field scanning options for them. Each different DFR grouping identifies a different document, or a largely different way of scanning the same document (for example, a document in a different font).

DFR and DLINT macros, after assembly and linkage editing, are preserved in loadable form until called for by the application program.

Each DLINT macro describes the scanning of a line, by field, in terms of

1. The starting and ending points of fields on a line (in tenths of an inch).
2. The field lengths (in characters).
3. The font code to be used (OCR-A, OCR-B, Gothic, or hand-printed numerics, all with various additional options).
4. Field editing (blank fill, blank suppression, zero fill, left or right justification, special character suppression).
5. Field character delimiters (a character to end a field scan).

Note that the DLINT macro may specify either standard mode or image mode. In standard mode, all DLINT options are valid, and the data record is of a fixed format, according to the field lengths in characters. In image mode, the field length and all EDIT keywords are invalid. The data record begins with 14 parameters, each two bytes long, indicating the length of the fields that follow. Because of this variable format in the data record, it is recommended that image mode be used only in applications for which standard mode is unsuitable.

COBOL Support

COBOL supports the OCR with a subprogram (invoked by CALL statements), Data Division COPY statement library material (to fully describe the parameter area required by the subprogram), and Procedure Division COPY statement library material (to provide procedures that simplify invocation of the subprogram).

File Description

The file is described by the Data Division COPY statement member. (See sample program for format.) All fields and codes are included, with descriptive names and default values. The programmer need only modify those fields that are not appropriate for the application.

The file description ("OCR-FILE" in the COPY statement member) includes all fields that the programmer must provide to the subprogram, the OCR-STATUS-KEY returned by the subprogram, and fields that describe the header and data records returned by the device. Note that the file is described through data records rather than the usual COBOL FD.

Note: The header and data records are not constructed under program control and are not altered after reading. Their contents are fully described in IBM 3886 Optical Character Reader General Information Manual.

Record Description

The COBOL record descriptions are based on the DLINT formats, either in image mode or in standard mode.

If standard mode scanning is specified, the data record is returned in a fixed format according to the DLINT macro; that is, contiguous fields, from left to right, in the same order as in the DLINT macro, each with a specified length in bytes. If image mode scanning is specified, however, the field lengths are returned at the beginning of the data record.

The programmer may describe the data records to be read by the application program by following the Data Division COPY statement request with statement(s) of the form:

```
     05  dataname REDEFINES OCR-DATA-RECORD
```

The structure of each record description should follow each such statement starting with a level number greater than 5. (See sample program for example.)

Procedural Code

The COBOL source statements control the file, read lines, and recover from errors. The subprogram CALL statement requirements are described in the Procedure Division COPY statement member. This member provides paragraphs which the COBOL programmer can PERFORM to set the proper operation code, CALL the subprogram, and pass control to a programmer-supplied exception routine if an exception occurs. The programmer should COPY these paragraphs into his program.

The programmer must move parameter information to the file area (OCR-FILE-CONTROL-AREA), and then issue a PERFORM statement for the appropriate procedure.

If an exception occurs, the COPY statement member passes control to the procedure-name OCR-EXCEPTION-ROUTINE. If operations are to be retried in this routine, the programmer should issue the appropriate CALL (not PERFORM) statement and test the OCR-STATUS-KEY value afterwards.

Return from the OCR-EXCEPTION-ROUTINE would normally be to OCR-CALL-EXIT (after a successful retry or recovery). Control is then returned to the invoking PERFORM statement.

JCL Considerations

Programs using the IBM-supplied 3886 processing subroutines must have a SIZE parameter specified on the EXEC card and cannot run in REAL mode. The user must specify the SIZE parameter equal to the size of his problem program to free the remainder of his partition for use as the page pool. Each opened 3886 file requires at least 2K bytes of the page pool.

Subprogram Interface

The IBM-supplied COPY members provide a data area ('OCR-FILE') and CALL statements using this area for parameter interface to the OCR subprogram. The data area has the following format:

```
01  OCR-FILE.
    05  OCR-FILE-CONTROL-AREA
        10  OCR-FILE-ID PIC X(8) VALUE
              'SYSnnn'.
            (Unique file name; also, must
              agree with JCL ASSGN
              statement)


        10  OCR-FORMAT-RECORD-ID PIC X(8)
              VALUE "xxxxxxxx".
            (DFR phase name, used for
              'OPEN' or "SETDV")

        10  OCR-OPERATION PIC X(5).
            ("OPEN", "CLOSE", "READ",
              "READO", "WAIT", "SETDV",
              "MARKL", "MARKD", or "EJECT"
              (left justified).

        10  OCR-STATUS-KEY PIC 99.
            (also referred to as exception
              code.)

        10  OCR-LINE

            15  OCR-LINE-NUMBER PIC 99.
                (Line number (0-33) passed
                  to "MARKL", "READ", or
                  "EJECT")

            15  OCR-LINE-FORMAT PIC 99.
                (Line format number (0-63)
                  passed to "READ")

        10  OCR-MARK PIC 99.
            (Mark option (1-15) passed to
              "MARKL" or "MARKD".)

        10  OCR-STACKER PIC 9.
            (Pocket number (1-2) passed to
              "EJECT".)

    05  OCR-HEADER-RECORD PIC X(20).
        (Header information returned from
          "READ" or "WAIT".)

    05  OCR-DATA-RECORD PIC X(130).
        (Data record returned from "READ"
          or "WAIT".)
```

(For descriptions of these operations, see the section "Statements for Invoking 3886 I/O Functions".)

Note: If the CALL statement does not have one, and only one, parameter following the USING option, the subprogram will return control immediately to the user (with a value of 8 in register 15). No error indication will be available through COBOL.

Table 37 contains OCR status key values and their meanings. Table 38 is a guide to which operations cause status key values 00 through 99. Table 39 supplies the user responses to status key values.

Table 37. OCR Status Key Values and User Actions

Status Key Code	Meaning
00	Successful completion
10	End-of-file[1]
3x	I/O error or related error where:[2] x = 1 -- Mark Check = 2 -- Nonrecovery = 3 -- Incomplete Scan = 4 -- Mark Check and Equipment Check = 9 -- Permanent Error
9y	Other error where: y = 2 -- logic error, that is, file not open (except OPEN), file already open (for OPEN), WAIT issued, but no READO pending, WAIT not issued for pending READO. = 3 -- insufficient storage available (OPEN) or failure in storage release (CLOSE) = 5 -- invalid parameter (other than operation code) = 9 -- unrecognizable operation code

[1]The end-of-file condition is raised after the listed I/O commands if:
-- the operator has pressed the END-OF-FILE button, and
-- no documents remain in the read station, and
-- no errors are outstanding

If // ASSGN SYSxxx,IGN has been specified, EOF is given only on READ and WAIT commands. While the end-of-file condition is active, commands (other than CLOSE) are only checked for validity.
[2]If any I-O errors, or certain system errors occur during the OPEN operation, the job is canceled by the system.

Table 38. Possible Status Key Values, By Operation

OCR-OPERATION Value										
OCR-STATUS-KEY Possible Value	OPEN	CLOSE	READ	READO	WAIT	MARKL	MARKD	EJECT	SETDV	other
00	X	X	X	X	X	X	X	X	X	
10			X		X	X	X	X	X	
31								X		
32			X	X	X	X	X	X	X	
33			X		X					
34								X		
39			X	X	X	X	X	X	X	
92	X	X	X	X	X	X	X	X	X	
93	X									
95	X		X	X		X	X	X		
99										X

Table 39. User Responses to Status Key

Status Key	Meaning	Response
00	Successful (no EOF)	The operation has completed properly.
10	End-of-file	Do EOF processing and close the file, or have operator ready 3886 and continue processing. See Note 1.
31	Mark Check	Attempt to reread the line, or eject document and prepare to process next document.
32	Nonrecovery Error	Eject document and prepare to process next document.
33	Incomplete Scan	Reread the line using a different DLINT, or using an image-mode DFR.
34	Mark Check and Equipment Check	See Note 2.
39	Permanent Error	See Note 2. One of the following has occurred: Command Reject, Bus Out Check, Equipment Check, Non-Initialized, RCP error, or Invalid Format.
92	Logic error	See Note 3. One of the following operation order errors has occurred: -- OPEN issued on file already open -- file not open (all operations except OPEN) -- WAIT issued but no READO in progress -- READO not followed by WAIT
93	Insufficient storage	See Note 3. The GETVIS issued by the COBOL subroutine has failed. Check that the SIZE parameter is large enough.
95	Invalid parameter	See Note 3. A parameter required by the last operation was invalid (too large, too small, or contained invalid characters).
99	Unrecognizable operation	See Note 3. The OCR-OPERATION parameter contained an illegal operation code.

Notes:

1. Serious I-O error conditions exist. No more I/O should be performed on the device after any of these errors are encountered. The program should indicate the error, perform error recovery, and issue a STOP RUN.

2. A serious programming error has occurred, or there is a problem in the program environment. The program should indicate the error, perform clean-up, and issue a STOP RUN.

3. WAIT and READ commands return data and header records only for the following codes: 00, 10, 31, and 33. For other codes, the contents of the header and data record areas are unpredictable.

STATEMENTS FOR INVOKING 3886 I/O FUNCTIONS

OPEN Function (Equivalent to OPEN Macro)

OPEN makes a logical file available to your program and loads the appropriate format record into the 3886. The statement format for OPEN is:

```
     PERFORM OCR-OPEN
```

The subprogram requires these fields: OCR-FILE-ID, OCR-OPERATION('OPEN'), OCR-FORMAT-RECORD-ID

The subprogram will return: OCR-STATUS-KEY

CLOSE Function (Equivalent to DOS CLOSE Macro)

CLOSE deactivates any 3886 files used by your program. These files must be closed before the program can be terminated. The statement format for CLOSE is:

```
     PERFORM OCR-CLOSE
```

The subprogram requires these fields: OCR-FILE-ID, OCR-OPERATION ('CLOSE')

The subprogram will return: OCR-STATUS-KEY

READ Function (Equivalent to DOS READ and WAITF Macros)

READ allows one line of data to be read from the document. The statement format for READ is:

```
     PERFORM OCR-READ
```

The subprogram requires these fields: OCR-FILE-ID, OCR-OPERATION ('READ'), OCR-LINE-NUMBER, OCR-LINE-FORMAT

The subprogram will return: OCR-STATUS-KEY, OCR-HEADER-RECORD, OCR-DATA-RECORD

Note: The READ function combines the functions of READO and WAIT. I/O overlap is not allowed within the issuing task.

READO Function (Equivalent to DOS READ Macro)

READO (read overlapped) initiates the reading of one line of data from the document. WAIT must subsequently be issued to complete the request. The statement format for READO is:

```
     PERFORM-OCR-READ-OVERLAPPED
```

The subprogram requires these fields: OCR-FILE-ID, OCR-OPERATION ('READO'), OCR-LINE-NUMBER, OCR-LINE-FORMAT

The subprogram will return: OCR-STATUS-KEY

Note: A successful READO function must be followed by a WAIT request for that same OCR-FILE area. No intervening I/O operations for that file are allowed.

WAIT Function (Equivalent to DOS WAITF Macro)

WAIT completes the action of the preceding READ. The statement format for WAIT is:

```
     PERFORM OCR-READ
```

The subprogram requires these fields: OCR-FILE-ID, OCR-OPERATION ('WAIT'),

The subprogram will return: OCR-STATUS-KEY, OCR-HEADER-RECORD, OCR-DATA-RECORD

The WAIT function causes the active task to be placed in the WAIT condition, if necessary, until the preceding READO operation is completed. It must be issued only after a successful READO, with no intervening commands for that file.

MARKL Function (Equivalent to DOS CNTRL Macro with LMK Option)

MARKL is used to mark a line on the document. The statement format for MARKL is:

```
     PERFORM OCR-MARK-LINE
```

The subprogram requires these fields: OCR-FILE-ID, OCR-OPERATION ('MARKL'), OCR-LINE-NUMBER, OCR-MARK

The subprogram will return: OCR-STATUS-KEY

MARKD Function (Equivalent to DOS CNTRL Macro with DMK Option)

MARKD is used to mark the document (in the Page Mark location).

The statement format for MARKD is:

```
     PERFORM OCR-MARK-DOCUMENT
```

The subprogram requires these fields: OCR-FILE-ID, OCR-OPERATION ('MARKD'), OCR-MARK

The subprogram will return: OCR-STATUS-KEY

EJECT Function (Equivalent to DOS CNTRL Macro, with ESP Option)

EJECT is used to eject the document into a specified stacker, with optional validation of its total number of timing marks. The statement format for EJECT is:

```
     PERFORM OCR-EJECT
```

The subprogram requires these fields: OCR-FILE-ID, OCR-OPERATION ('EJECT'), OCR-STACKER, OCR-LINE-NUMBER

The subprogram will return: OCR-STATUS-KEY

SETDV (Set Device by Loading a Format Record) Function (Equivalent to DOS SETDEV Macro)

SETDV allows format records to be changed during execution of the program. The statement format for SETDV is:

```
     PERFORM OCR-SET-DEVICE
```

The subprogram requires these fields: OCR-FILE-ID, OCR-OPERATION ('SETDV'), OCR-FORMAT-RECORD-ID

The subprogram will return: OCR-STATUS-KEY

COBOL 3886 Library Routine

The COBOL 3886 library routine is invoked in response to the CALL statement. For the proper execution of this routine GETVIS=YES must be specified at system generation. An illegal SVC results if GETVIS=NO is specified.

Table 40 contains a list of CALL statements used for invoking 3886 I/O functions (if the IBM-supplied COPY member is not used).

All OCR CALL statements have the format CALL 'ILBDOCR0' USING OCR-FILE, where OCR-FILE is used as follows:

Table 40. CALL Statements for Invoking 3886 I/O Functions

Function (OCR-OPERATION)	Set by User	Subroutine Returns
OPEN	OCR-FILE-ID OCR-OPERATION OCR-FORMAT-RECORD-ID	OCR-STATUS-KEY
CLOSE	OCR-FILE-ID OCR-OPERATION	OCR-STATUS-KEY
READ	OCR-FILE-ID OCR-OPERATION OCR-LINE-NUMBER OCR-LINE-FORMAT	OCR-STATUS-KEY OCR-HEADER-RECORD OCR-DATA-RECORD
READO	OCR-FILE-ID OCR-OPERATION OCR-LINE-NUMBER OCR-LINE-FORMAT	OCR-STATUS-KEY
WAIT	OCR-FILE-ID OCR-OPERATION	OCR-STATUS-KEY OCR-HEADER-RECORD OCR-DATA-RECORD
MARKL	OCR-FILE-ID OCR-OPERATION OCR-LINE-NUMBER OCR-MARK	OCR-STATUS-KEY
MARKD	OCR-FILE-ID OCR-OPERATION OCR-LINE-NUMBER OCR-MARK	OCR-STATUS-KEY
EJECT	OCR-FILE-ID OCR-OPERATION OCR-LINE-NUMBER OCR-STACKER	OCR-STATUS-KEY
SETDV	OCR-FILE-ID OCR-FORMAT-RECORD-ID OCR-OPERATION	OCR-STATUS-KEY

PROCESSING TAPES FROM THE OCR 3886, MODEL 2

Tape records produced from the IBM 3886, Model 2 are almost identical in format to the header and data records returned by the Model 1. The main differences between the records are:

- Model 2 tapes contain a document trailer record after the line output records for each document. The content of this trailer record differs from that of line output records.
- The codes used in certain fields of the header record differ between the two models.

Because of the similarity, however, the Data Division COPY statement member defined for the Model 1 may be tailored to describe the Model 2 tape records. To do this, punch out the COPY statement member, modify it according to the installation requirements, and recatalog it. The COPY statement member may then be included as a data record, under an FD for the input tape file.

Specific information on the formats and contents of the Model 2 tape records is contained in _IBM 3886 Optical Character Reader, General Information Manual_.

```
******************************************************************91547000
*********     S A M P L E   O C R   P R O G R A M        *********91548000
******************************************************************91549000
 IDENTIFICATION DIVISION                                          91550000
   PROGRAM-ID. SAMPLE
****** THIS PROGRAM IS THE COBOL EQUIVALENT OF THE                91551200
*        ASSEMBLY LANGUAGE SAMPLE PROGRAM 'DOCLIST',              91551400
*        CONTAINED IN THE DOS/VS PROGRAM PLANNING GUIDE           91551600
*        FOR THE IBM 3886 OPTICAL CHARACTER READER, MODEL 1       91551800
*        (ORDER NO. GC21-5059)                                    91551900
 ENVIRONMENT DIVISION.                                            91552000
 INPUT-OUTPUT SECTION.                                            91552200
 FILE-CONTROL.                                                    91552400
     SELECT PRINTER, ASSIGN TO SYS009-UR-1403-S.                  91552600
 DATA DIVISION.                                                   91553000
 FILE SECTION.                                                    91553200
 FD  PRINTER LABEL RECORDS ARE OMITTED.                           91553400
 01  PRINT-RECORD.                                                91553600
     05  FILLER          PIC X.                                   91553800
     05  PRINT-LINE      PIC X(130).                              91553900
 WORKING-STORAGE SECTION.                                         91554000
 77  PRINT-CONTROL               PIC 9       VALUE 1.             91554200
 77  MSG-PERMANENT-ERROR         PIC X(24)   VALUE                91555000
         'PERMANENT ERROR OCCURRED'.                              91556000
 77  MSG-MARK-CHECK              PIC X(19)   VALUE                91556100
         'MARK CHECK OCCURRED                                     91556200
 77  MSG-MARK-AND-EQUIP-CHECK    PIC X(39)   VALUE                91556600
         'MARK CHECK AND EQUIPMENT CHECK OCCURRED'.               91556700
 77  MSG-INCOMPLETE-SCAN         PIC X(24)   VALUE                91556800
         'INCOMPLETE SCAN OCCURRED'.                              91556900
 77  MSG-NONRECOVERY-ERROR       PIC X(26)   VALUE                91557000
         'NONRECOVERY ERROR OCCURRED'.                            91557200
 77  MSG-BAD-DATA                PIC X(50)   VALUE                91557400
         'THE FOLLOWING LINE WAS MISREAD.  THE LINE HEADER ='.    91557600
 01  MSG-TERMINATION.                                             91557700
     05  FILLER                  PIC X(44)   VALUE                91557800
         'TERMINAL ERROR OCCURRED - OCR-STATUS-KEY = '.           91558100
     05  MSG-TERM-STATUS-KEY     PIC XX.                          91558300
 01  OCR-FILE    COPY ILBDOCRD.                                   91558400
******** ILBDOCRD - OCR DATA DESCRIPTION ************************
```

Figure 69. Sample OCR Program (Part 1 of 5)

```
******************************************************************90037000
********   O C R   3 8 8 6   F I L E   F O R M A T      **********90047000
******************************************************************90057000
 01  OCR-FILE.                                                    90067000
     05  OCR-FILE-CONTROL-AREA.                                   90069000
         10  OCR-FILE-ID             PIC X(8)    VALUE 'SYS010  '.90077000
         10  OCR-FORMAT-RECORD-ID    PIC X(8)    VALUE 'FRLGDFR1'.90087000
         10  OCR-OPERATION           PIC X(5)    VALUE 'OPEN '.   90097000
             88  OCRO-OPEN                       VALUE 'OPEN '.   90107000
             88  OCRO-CLOSE                      VALUE 'CLOSE'.   90117000
             88  OCRO-READ                       VALUE 'READ '.   90127000
             88  OCRO-READ-OVERLAPPED            VALUE 'READO'.   90137000
             88  OCRO-WAIT                       VALUE 'WAIT '.   90147000
             88  OCRO-MARK-LINE                  VALUE 'MARKL'.   90157000
             88  OCRO-MARK-DOCUMENT              VALUE 'MARKD'.   90167000
             88  OCRO-EJECT                      VALUE 'EJECT'.   90177000
             88  OCRO-SETDEV                     VALUE 'SETDV'.   90187000
         10  OCR-STATUS-KEY          PIC 99      VALUE 0.         90197000
             88  OCRS-SUCCESSFUL                 VALUE 00.        90217000
             88  OCRS-END-OF-FILE                VALUE 10.        90227000
             88  OCRS-IO-ERRORS                  VALUE 30 THRU 39.90257000
             88  OCRS-MISC-ERROR                 VALUE 30.        90267000
             88  OCRS-MARK-CHECK                 VALUE 31.        90277000
             88  OCRS-NONRECOVERY-ERROR          VALUE 32.        90287000
             88  OCRS-INCOMPLETE-SCAN            VALUE 33.        90297000
             88  OCRS-MARK-AND-EQUIP-CHECK       VALUE 34.        90307000
             88  OCRS-PERMANENT-ERROR            VALUE 39.        90317000
             88  OCRS-SPECIAL-ERRORS             VALUE 90 THRU 99.90317400
             88  OCRS-LOGIC-ERROR                VALUE 92.        90323000
             88  OCRS-RESOURCE-UNAVAILABLE       VALUE 93.        90325000
             88  OCRS-INVALID-PARAMETER          VALUE 95.        90326000
             88  OCRS-INVALID-OPERATION          VALUE 99.        90326200
         10  OCR-LINE.                                            90327000
             15  OCR-LINE-NUMBER     PIC 99      VALUE 1.         90337000
             15  OCR-LINE-FORMAT     PIC 99      VALUE 1.         90347000
         10  OCR-MARK                PIC 99      VALUE 0.         90357000
         10  OCR-STACKER             PIC  9      VALUE 1.         90367000
*                                                                 90377000
*    ******* HEADER AND DATA RECORD AREAS *******                 90387000
*    FILLED IN BY SUCCESSFUL 'READ' AND/OR 'WAIT'.                90397000
*    (NOTE - 'READO' DOES NOT ALTER THESE AREAS)                  90407000
*                                                                 90417000
     05  OCR-HEADER-RECORD                       VALUE ZEROS.     90427000
         10  OCRH-LINE-NUMBER        PIC 99.                      90437000
         10  OCRH-LINE-FORMAT        PIC 99.                      90447000
         10  OCRH-LINE-SCAN-COUNT    PIC  9.                      90457000
         10  OCRH-LINE-STATUS        PIC  9.                      90467000
             88  OCRH-LINE-GOOD                  VALUE 0.         90477000
             88  OCRH-LINE-BLANK                 VALUE 1.         90487000
             88  OCRH-LINE-GROUP-ERASE           VALUE 3.         90497000
             88  OCRH-LINE-CRITICAL-ERR          VALUE 2.         90507000
             88  OCRH-LINE-NON-CRITICAL-ERR      VALUE 4.         90517000
             88  OCRH-LINE-COMBINED-ERR          VALUE 6.         90527000
             88  OCRH-LINE-INVALID               VALUE 7.         90537000
             88  OCRH-END-OF-PAGE                VALUE 5.         90547000
         10  OCRH-FIELD-INFO.                                     90557000
             15  OCRH-FIELD-STATUS   PIC 9   OCCURS 14.           90567000
                 88  OCRH-FIELD-GOOD             VALUE 0.         90577000
                 88  OCRH-FIELD-REJECT-CHARS     VALUE 2.         90587000
                 88  OCRH-FIELD-WRONG-LENGTH     VALUE 4.         90597000
                 88  OCRH-FIELD-COMBINED-ERR     VALUE 6.         90607000
                 88  OCRH-FIELD-BLANK            VALUE 8.         90617000
                 88  OCRH-FIELD-BLANK-SUP        VALUE 4.         90627000
     05  OCR-DATA-RECORD.                                         90637000
         10  OCR-STANDARD-MODE-RECORD                             90647000
             15  OCR-STANDARD-FIELD-CHAR PIC X   OCCURS 130.      90657000
         10  OCR-IMAGE-MODE-RECORD                                90667000
                     REDEFINES OCR-STANDARD-MODE-RECORD.          90677000
             15  OCR-IMAGE-FIELD-LENGTH  PIC 99  OCCURS 14.       90687000
             15  OCR-IMAGE-FIELD-CHAR    PIC X   OCCURS 102.      90697000
********** END OF 3886 DATA DIVISION COPY MEMBER *************    90699000
     05  NOTICE-OF-PAYMENT-DUE REDEFINES OCR-DATA-RECORD.         91561400
         10  LINE-1.                                              91561600
             15  L1-POLICYHOLDER-NAME    PIC X(20).               91561800
             15  FILLER PIC X(15).
         10  LINE-2 REDEFI ES LINE-1.                             91561900
             15  L2-CITY-AND-STATE       PIC X(20).               91562200
             15  L2-POLICY-NUMBER        PIC X(8).                91562400
             15  L2-AMOUNT-DUE           PIC 9(4)V99.             91562600
             15  L2-PAYMENT-VERIFY-CODE  PIC 9.                   91562700
         10  LINE-3 REDEFINES LINE-1.                             91562800
             15  L3-AMOUNT-PAID      PIC 9(5)V99.                 91563100
```

Figure 69. Sample OCR Program (Part 2 of 5)

```
 PROCEDURE DIVISION.                                              91563400
     STOP RUN.                                                    91563700
 P10-START.                                                       91564000
     MOVE 'SYS010' TO OCR-FILE-ID.                                91565000
     MOVE 'FORMAT' TO OCR-FORMAT-RECORD-ID.                       91566000
     PERFORM OCR-OPEN.                                            91567000
     OPEN OUTPUT PRINTER.                                         91568000
 P10-HEAD.                                                        91569000
     MOVE ALL '*' TO PRINT-LINE.                                  91570000
     PERFORM PRINT-ROUTINE.                                       91571000
     MOVE 1 TO OCR-STACKER.                                       91572000
 P10-READ.                                                        91573000
     PERFORM OCR-READ.                                            91574000
     IF OCRS-NONRECOVERY-ERROR,  GO TO P10-EOP-ERR.               91574200
     IF OCRH-LINE-GOOD,          GO TO P10-GOOD.                  91575000
     IF OCRH-LINE-BLANK,         GO TO P10-GOOD.                  91576000
     IF OCRH-LINE-NON-CRITICAL-ERR, GO TO P10-GOOD.               91579000
     IF OCRH-END-OF-PAGE,        GO TO P10-EOP.                   91580000
***** IF OCRH HAS ANY OTHER CODE, CONSIDER THE DATA AS BAD ****   91581000
 P10-BAD.                                                         91582000
     MOVE MSG-BAD-DATA TO PRINT-LINE.                             91583000
     PERFORM PRINT-ROUTINE.                                       91584000
     MOVE 2 TO OCR-STACKER.                                       91584200
 P10-GOOD.                                                        91585000
     MOVE OCR-DATA-RECORD TO PRINT-LINE.                          91585200
     PERFORM PRINT-ROUTINE.                                       91585400
     MOVE 1 TO PRINT-CONTROL.                                     91585600
     ADD 1 TO OCR-LINE-NUMBER, OCR-LINE-FORMAT.                   91585800
     IF OCRH-LINE-NUMBER IS LESS THAN 3, GO TO P10-READ.          91585900
 P10-EOP.                                                         91586200
     MOVE 3 TO OCR-LINE-NUMBER.                                   91586300
     PERFORM OCR-EJECT.                                           91586400
 P10-EOP-ERR.                                                     91586600
     MOVE 1 TO OCR-LINE-NUMBER, OCR-LINE-FORMAT.                  91586700
     MOVE 3 TO PRINT-CONTROL.                                     91586800
     GO TO P10-HEAD.                                              91587100
********* EXCEPTION PROCESSING ROUTINE ***************             91587300
 OCR-EXCEPTION-ROUTINE.                                           91587400
     IF OCRS-END-OF-FILE,    GO TO P20-EOF.                       91587600
     IF OCRS-MARK-CHECK,                                          91587700
         MOVE MSG-MARK-CHECK TO PRINT-LINE,                       91587800
         GO TO P20-RETURN.                                        91587900
     IF OCRS-NONRECOVERY-ERROR,                                   91588100
         MOVE MSG-NONRECOVERY-ERROR TO PRINT-LINE,                91588500
         GO TO P20-RETURN.                                        91588700
     IF OCRS-INCOMPLETE-SCAN,                                     91588900
         MOVE MSG-INCOMPLETE-SCAN TO PRINT-LINE,                  91589100
         GO TO P20-RETURN.                                        91589300
     IF OCRS-MARK-AND-EQUIP-CHECK,                                91589500
         MOVE MSG-MARK-AND-EQUIP-CHECK TO OCR-LINE,               91589700
         GO TO P20-PRINT-EOF.                                     91590000
     IF OCRS-PERMANENT-ERROR,                                     91591000
         MOVE MSG-PERMANENT-ERROR TO PRINT-LINE,                  91592000
         GO TO P20-PRINT-EOF.                                     91593000
***** IF NONE OF THE ABOVE ERRORS, GIVE TERMINATION MESSAGE ***** 91594000
     MOVE OCR-STATUS-KEY TO MSG-TERM-STATUS-KEY.                  91595000
     MOVE MSG-TERMINATION TO PRINT-LINE.                          91596000
     GO TO P20-PRINT-EOF.                                         91597000
 P20-RETURN.                                                      91598000
     PERFORM PRINT-ROUTINE.                                       91598200
     GO TO OCR-CALL-EXIT.                                         91598400
 P20-PRINT-EOF.                                                   91600000
     PERFORM PRINT-ROUTINE.                                       91601000
 P20-EOF.                                                         91602000
     PERFORM OCR-CLOSE.                                           91603000
     CLOSE PRINTER.                                               91604000
     STOP RUN.                                                    91605000
 PRINT-ROUTINE.                                                   91607000
     WRITE PRINT-RECORD AFTER ADVANCING PRINT-CONTROL.            91609000
 OCR-COPIED-PROCEDURES.  COPY ILBDOCRP.                           91610000
```

Figure 69. Sample OCR Program (Part 3 of 5)

```
******* ILBDOCRP - OCR 3886 PROCEDURES
*****************************************************************90757000
********    O C R   3 8 8 6   P R O C E D U R E S      *********90767000
*****************************************************************90777000
*  THE 3886 OCR SUBROUTINE USES OCR-FILE FIELDS AS FOLLOWS        90778000
*                                                                 90779000
*  ALL OPERATIONS REQUIRE                                         90780000
*      OCR-FILE-ID = THE UNIQUE NAME USED TO IDENTIFY THE FILE    90781000
*                            TO THE SUBROUTINE AND TO THE SYSTEM  90782000
*      OCR-OPERATION = THE CODE FOR THE REQUESTED OPERATION       90783000
*  ALL OPERATIONS RETURN                                          90784000
*      OCR-STATUS-KEY = RETURN CODE FOR VARIOUS OCCURRENCES       90785000
*                                                                 90786000
*  OCR-OPEN ('OPEN ') ALSO REQUIRES                               90786200
*      OCR-FORMAT-RECORD-ID = LIBRARY NAME OF DFR TO LOAD         90786400
*  OCR-CLOSE ('CLOSE') REQUIRES NO ADDITIONAL PARAMETERS          90786600
*  OCR-READ ('READ ') ALSO REQUIRES                               90786800
*      OCR-LINE-NUMBER (1-33) = LINE TO READ (ON DOCUMENT)        90786900
*      OCR-LINE-FORMAT (1-63) = DLINT NUMBER (IN CURRENT DFR)     90787900
*    AND RETURNS (IF OCRS-SUCCESSFUL)                             90788100
*      OCR-HEADER-RECORD = HEADER RECORD, AS RETURNED BY THE 3886 90788300
*      OCR-DATA-RECORD = DATA FROM DOCUMENT, FROM 3886            90788500
*  OCR-READ-OVERLAPPED ('READO') HAS SAME REQUIREMENTS AS OCR-READ90788800
*  OCR-WAIT ('WAIT ') RETURNS SAME PARAMETERS AS OCR-READ         90789800
*  OCR-MARK-LINE ('MARKL') ALSO REQUIRES                          90790000
*      OCR-LINE-NUMBER (1-33) = LINE TO MARK (ON DOCUMENT)        90790200
*      OCR-MARK (1-15) = SUM OF DESIRED MARK CODES (8421)         90790400
*  OCR-MARK-DOCUMENT ('MARKD') ALSO REQUIRES                      90790600
*      OCR-MARK (1-15) = SUM OF DESIRED MARK CODES (8421)         90790700
*  OCR-EJECT ('EJECT') ALSO REQUIRES                              90791700
*      OCR-STACKER (1-2) = STACKER TO SELECT (A OR B)             90791900
*      OCR-LINE-NUMBER (0-33) = NUMBER OF LINES ON DOCUMENT       90792100
*                FOR VALIDATION (IF 0, NO VALIDATION WILL OCCUR)  90792500
*  OCR-SET-DEVICE ('SETDV') ALSO REQUIRES                         90792600
*      OCR-FORMAT-RECORD-ID = LIBRARY NAME OF DFR TO LOAD         90793600
*                                                                 90793800
*NOTES-                                                           90794000
*  1.  THE TERMS DFR AND DLINT ARE USED TO REFER TO THE EXPANDED  90794200
*  CODE, IN LOADABLE FORM, OF THE RESPECTIVE SYSTEM MACROS.       90794400
*  2.  OCR-WAIT MAY BE REQUESTED AFTER, AND ONLY AFTER, A         90795300
*  SUCCESSFUL OCR-READ-OVERLAPPED REQUEST.  NO INTERVENING        90795500
*  I/O COMMANDS WILL BE ALLOWED ON THAT SAME FILE.                90795700
*  3.  THE PROCEDURES PROVIDED BELOW AUTOMATICALLY FILL IN        90795900
*  THE OCR-OPERATION FIELD, CALL THE SUBROUTINE, AND TEST         90796100
*  THE OCR-STATUS-KEY AFTER RETURN.  IF ANY EXCEPTIONAL           90796400
*  CONDITIONS OCCUR, THEY PASS CONTROL TO THE ROUTINE             90796600
*  OCR-EXCEPTION-ROUTINE, WHICH THE PROGRAMMER MUST PROVIDE.      90796700
*  THE PROGRAMMER MAY AVOID EXCEPTION ROUTINE INVOCATION BY       90797900
*  ADDING THE FOLLOWING PHRASE TO THE COPY STATEMENT:             90798100
*         REPLACING OCR-EXCEPTION-ROUTINE BY OCR-CALL-EXIT        90798300
*  4.  ALTHOUGH OCR-STATUS-KEY MAY INDICATE THAT THE DESIRED
*  OPERATION WAS SUCCESSFUL, THE VALIDITY OF THE DATA OBTAINED
*  SHOULD BE DETERMINED BY TESTING OCRH-LINE-STATUS.
*****************************************************************90798700
```

Figure 69. Sample OCR Program (Part 4 of 5)

```
OCR-3886-PROCEDURES.                                              90799700
OCR-OPEN.                                                         90800700
    MOVE 'OPEN ' TO OCR-OPERATION OF OCR-FILE.                    90807000
    PERFORM OCR-CALL THRU OCR-CALL-EXIT.                          90817000
OCR-CLOSE.                                                        90827000
    MOVE 'CLOSE' TO OCR-OPERATION OF OCR-FILE.                    90837000
    PERFORM OCR-CALL THRU OCR-CALL-EXIT.                          90847000
OCR-READ.                                                         90857000
    MOVE 'READ ' TO OCR-OPERATION OF OCR-FILE.                    90867000
    PERFORM OCR-CALL THRU OCR-CALL-EXIT.                          90877000
OCR-READ-OVERLAPPED.                                              90887000
    MOVE 'READO' TO OCR-OPERATION OF OCR-FILE.                    90897000
    PERFORM OCR-CALL THRU OCR-CALL-EXIT.                          90907000
OCR-WAIT.                                                         90917000
    MOVE 'WAIT ' TO OCR-OPERATION OF OCR-FILE.                    90927000
    PERFORM OCR-CALL THRU OCR-CALL-EXIT.                          90937000
OCR-MARK-LINE.                                                    90947000
    MOVE 'MARKL' TO OCR-OPERATION OF OCR-FILE.                    90957000
    PERFORM OCR-CALL THRU OCR-CALL-EXIT.                          90967000
OCR-MARK-DOCUMENT.                                                90977000
    MOVE 'MARKD' TO OCR-OPERATION OF OCR-FILE.                    90987000
    PERFORM OCR-CALL THRU OCR-CALL-EXIT.                          90997000
OCR-EJECT.                                                        91007000
    MOVE 'EJECT' TO OCR-OPERATION OF OCR-FILE.                    91017000
    PERFORM OCR-CALL THRU OCR-CALL-EXIT.                          91027000
OCR-SET-DEVICE.                                                   91037000
    MOVE 'SETDV' TO OCR-OPERATION OF OCR-FILE.                    91047000
    PERFORM OCR-CALL THRU OCR-CALL-EXIT.                          91057000
OCR-CALL.                                                         91067000
    CALL 'ILBDOCR0' USING OCR-FILE.                               91077000
    IF NOT OCRS-SUCCESSFUL OF OCR-FILE,                           91087000
        GO TO OCR-EXCEPTION-ROUTINE.                              91097000
OCR-CALL-EXIT. EXIT.                                              91107000
********** END OF 3886 PROCEDURE DIVISION COPY MEMBER *********   91109000
```

Figure 69. Sample OCR Program (Part 5 of 5)

This index is supplemented with entries from the index of IBM DOS Full American National Standard COBOL. These entries are identified by an asterisk (*).

(Where more than one page reference is given, the major reference appears first.)

SC28-6478-3

IBM DOS/VS COBOL P.G. Printed in U.S.A. SC28-6478-3

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, N.Y. 10604

IBM World Trade Americas/Far East Corporation
Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591

IBM World Trade Europe/Middle East/Africa Corporation
360 Hamilton Avenue, White Plains, N.Y., U.S.A. 10601